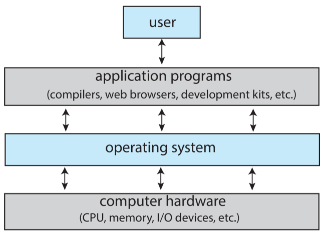
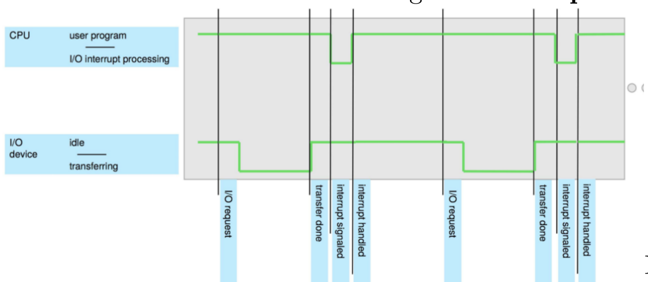
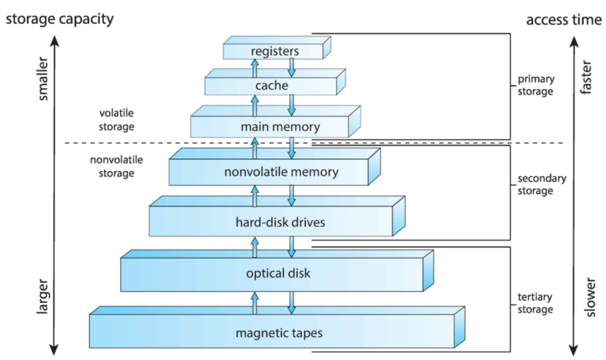
1 Operating System

OS = coordiantes use of hardware among various applications and users.



I/O devices and CPU can execute concurrently. The devices communicate with the CPU through **interrupts**.



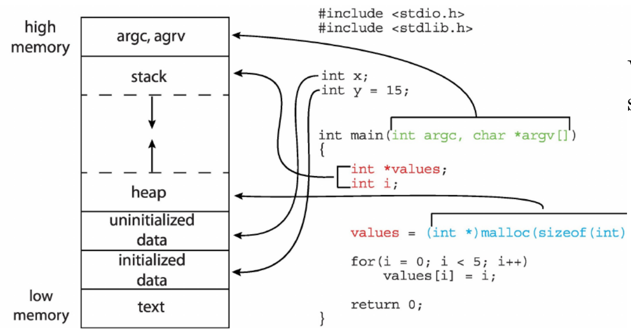


2 Processes

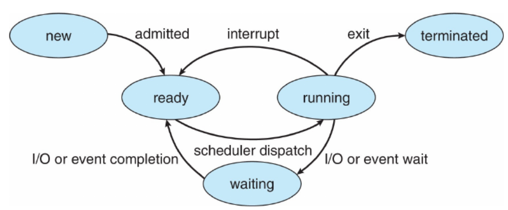
2.1 General

Process = program (executable / binary) in execution.
Program = passive entity stored on disk; Process = active entity loaded in memory.
Parts of a process:

- Text section (program code)
- Current activity (program counter and process registers)
- Stack (function parameters, local vars, ret adresses)
- Data section (global variables)
- Heap (dynamically alocated memory)



Process states: New, Running, Waiting, Ready, Terminated.

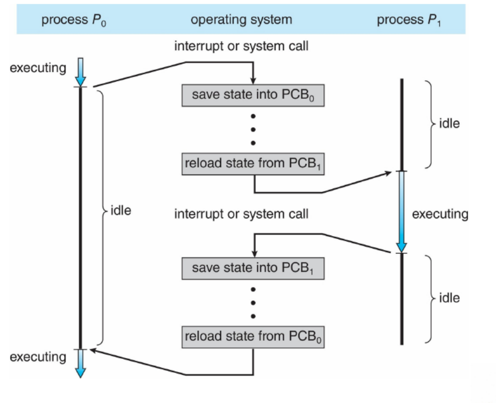


2.2 PCB

Information associated with each process (also called task control block)
PCB contents:

- Process state – running, waiting, etc.
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files

When CPU switches context, the respective state stored in the PCB is loaded:



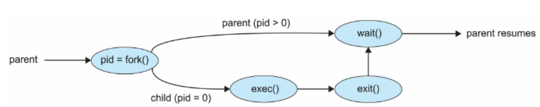
2.3 Process creation / termination

Parent process create children processes, which, in turn create other processes, forming a tree of processes.
Process is identified and managed via a process identifier (pid).

- fork() = syscall to create a new child process. Child's adress space is the same as parent. Returns 0 in the child process and PID of the child in parent.
- exec() = replace the child's adress space with another program.
- wait() = parent process calls wait() to wait for the child to terminate.

- exit() = process executes last statement and then asks the operating system to delete it.
- abort() = terminate the execution of children processes.
- waitpid() = waits for a specific child spawned by the process.

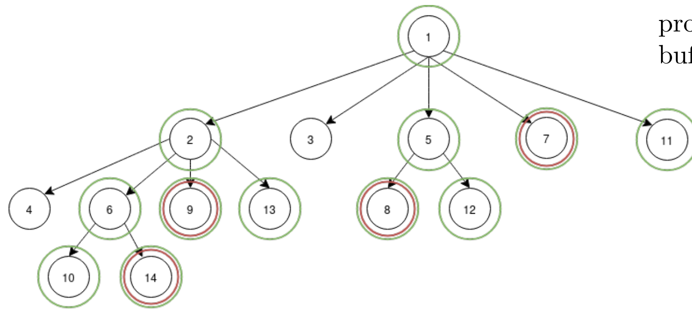
```
int main(int argc, char *argv[])
{
    pid_t pid = fork();
    if(pid < 0)
        return errno;
    else if(pid == 0)
    {
        //child
        char *argv[] = {"ls", NULL};
        execve("/bin/ls", argv, NULL);
        perror(NULL);
    }
    else
    {
        //parent
        wait(NULL);
        // child finished
    }
    return 0;
}
```



Exemplu arbore de procese: Câte procese și thread-uri sunt la final? Desenați arborescența de procese și thread-urile aferente.

```
fork()
if (fork()) {
    fork()
    if (!fork())
        pthread_create()
    else
        fork()
    pthread_create()
}
```

Sunt 14 procese si 16 thread-uri. Fiecare cerc este un thread format (prima data cele rosii, la linia 5, apoi cele verzi la linia 8).

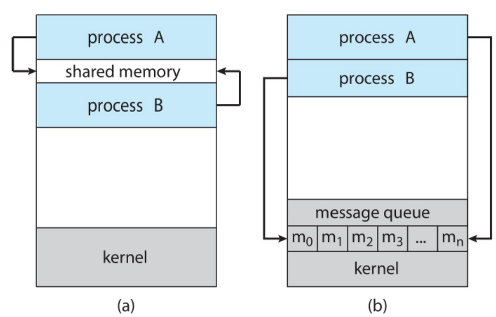


Zombie and Orphan processes:

- Zombie process = a process that has completed execution but still has an entry in the system process table. This happens because the parent didn't invoke wait().
- Orphan process = a living child whose parent has died. Could be because parent exited prematurely or intended in the case of daemons. The process is adopted by process with PID=1.

2.4 Interprocess Communication

Processes within a system may be independent or co-operating. Cooperating processes can affect or be affected by other processes for: sharing data, computation speedup, modularity, convenience.
Two models of IPC: Shared memory and Message passing.



2.4.1 Shared memory

Requires careful synchronization.
Producer-Consumer problem: producer process produces information that is consumed by a consumer process. Solved with shared memory by holding a buffer and in/out pointers.

- unbounded-buffer = Producer never waits; Consumer waits if there is no buffer to consume
- bounded-buffer = Producer must wait if all buffers are full; Consumer waits if there is no buffer to consume

Shared memory used in UNIX with shm_open() to create a shared memory segment and mapped it to a file descriptor via mmap.

2.4.2 Message passing

Processes communicate with each other without resorting to shared variables.
IPC facility provides two operations: **send(message)** and **receive(message)**

- Direct Communication = Processes must name each other explicitly: send (P, message) - send a message to process P, receive(Q, message) - receive a message from process Q
- Indirect Communication = Send and receive messages through mailbox: send(A, message) - send a message to mailbox A, receive(A, message) - receive a message from mailbox A

Message passing may be either blocking or non-blocking:

- Blocking (synchronous)
  - Blocking send = the sender is blocked until the message is received
  - Blocking receive = the receiver is blocked until a message is available
- Non-blocking (asynchronous)
  - Non-blocking send = the sender sends the message and continue
  - Non-blocking receive = the receiver receives a valid message / NULL

If both send and receive are blocking, we have a **rendezvous**.
As processes can not be synchronized perfectly, we make use of buffering (queue of messages attached to the link):

- Zero capacity: no messages are queued on a link. Sender must wait for receiver. (rendezvous)
- Bounded capacity: finite length of n messages. Sender must wait if link full.
- Unbounded capacity: infinite length. Sender never waits.

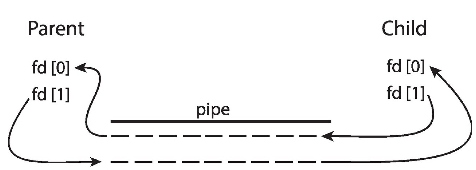
2.4.3 Signals

Signals can be send to a process:

- SIGINT (ctrl+c) = kills the process
- SIGSTOP (ctrl+z) = stops the process, moves it to background
- SIGSEGV = invalid memory accesed
- etc.

2.4.4 Pipes

Acts as a conduit allowing two processes to communicate.
Ordinary pipes (anonymous) = cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
Named pipes = can be accessed without a parent-child relationship.



2.4.5 Sockets

A socket is defined as an endpoint for communication. Concatenation of **IP address and port** (a number included at start of message packet to differentiate network services on a host).
loopback (127.0.0.1) = system on which process is running

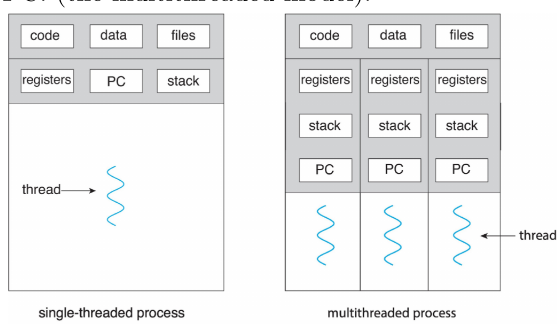
2.4.6 RPCs

RPCs = remote procedure calls. Abstracts procedure calls between processes on networked systems Stubs = client-side proxy for the actual procedure on the server The client-side stub locates the server and marshalls the parameters. The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server

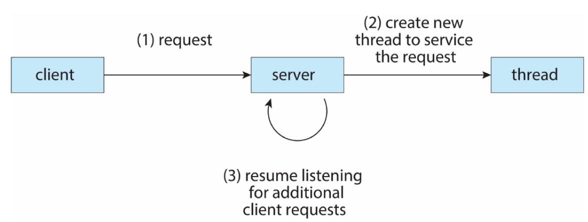
3 Threads

General

A thread is the basic unit of CPU utilization. It is less costly than process, because each process can have multiple threads that share resources (data, code, files), having only independent registers, stack and PC. (the multithreaded model).

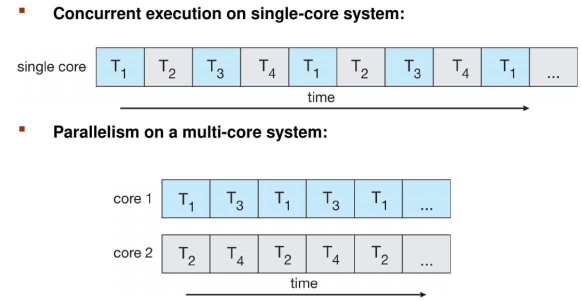


They can be used in the server model to process requests

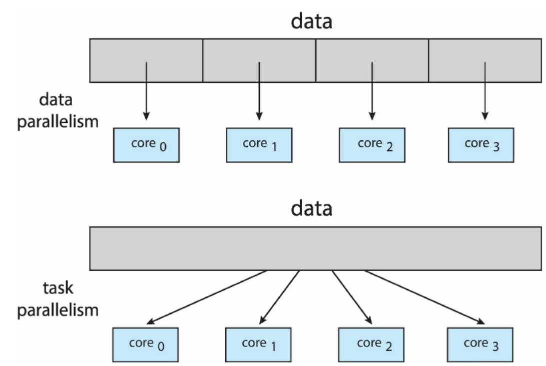


Parallelism & concurrency

- **Parallelism** = a system can execute more than one task simultaneously.
- **Concurrency** = more than one task is making progress.



- **data parallelism** = distribute subsets of same data across multiple cores, same operation
- **task parallelism** = distribute threads across multiple cores, each with unique operation



Amdahl's law

This measures the performance gain from adding cores to an application that has both sequential and parallel components.
- S - serial portion (fraction), $1 - S$ - parallel portion
- N - cores

$$speedup \leq \frac{1}{S + \frac{1-S}{N}}$$

Applied. $S = 0.25$ (25% serial), moving from 1 to 2 cores $N = 2$.

$$s \leq \frac{1}{0.25 + \frac{0.75}{2}} = 1.6$$

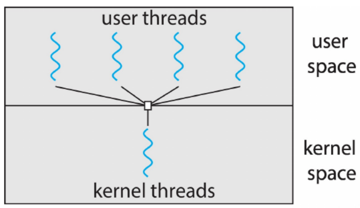
Speedup of up to 1.6 times.

User vs Kernel Threads

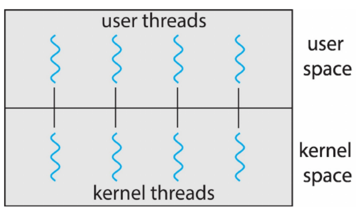
- **User thread** = managed by user level library (e.g. pthread)
- **Kernel Thread** = supported by the Kernel

There are three models that map user threads to kernel threads.

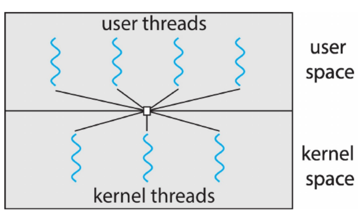
- **many-to-one**: many user threads mapped to 1 kernel thread. One thread blocking causes others to block.



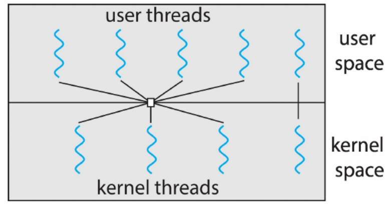
- **one-to-one**: 1 user threads mapped to 1 kernel thread. More concurrency than previous.



- **many-to-many**: many user threads mapped to many kernel threads. Allows OS to create sufficient number of kernel threads.



- **two level**: similar to M:M but allows one user thread to be bound to kernel thread.



Other

Implicit threading represents the growing trend of having threads created and managed by compilers and run-time libraries, instead of creating them explicitly. Many methods.

- thread pools : create a pool of threads that await work
- fork-join parallelism: like divide et impera, but on threads

```
Task(problem)
   if problem is small enough
      solve the problem directly
   else
      subtask1 = fork(new Task(subset of problem)
      subtask2 = fork(new Task(subset of problem)

      result1 = join(subtask1)
      result2 = join(subtask2)

      return combined results
```

- OpenMP - set of compiler directives that identify parallel regions `#pragma omp parallel`
- there exists a complication on **fork** and **exec**. - there exists some versions of fork (one that forks all threads, one that forks only the thread that calls the fork) - exec is simpler and it replaces all the threads with the new code.

Signal

Signals are used in UNIX systems to notify a process that something has happened.

A signal handler is used to process a signal. We have predefined and user defined handlers.

A signal can be delivered to all threads, to certain threads or the thread that the signal applies to.

- **thread cancellations** is the termination of a thread before it finishes. There are two approaches **asynchronous cancellation** - terminate immediately and **deffered cancellation** allows the thread to check periodcally if it should be cancelled.

Cancellation can be disabled by the thread.

- **Thread Local Storage** - each thread can have its own copy of data

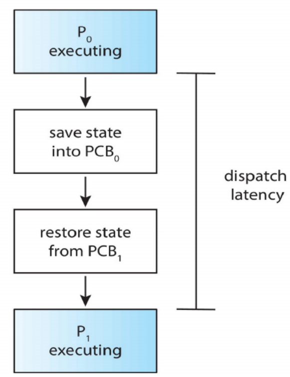
4 CPU Scheduling

How to schedule processes on the CPU efficently? Usually, CPU is a sequence of CPU bursts and I/O bursts (waiting time for I/O). A CPU scheduler selects a process and allocates a core to it.

- **Preemptive** = can postpone processes (put process in ready state / back in running state)
- **Non-preemptive** = once process has CPU, the process keeps it until the end or by switching to waiting.

Dispatcher = gives control of the CPU to the process selected by the CPU scheduler: switching context, switching to user mode, jumping to proper location in the user program to restart it.

Dispatch latency = time it takes for the dispatcher to stop one process and start another running.



Scheduling criteria:

- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes that complete their execution per time unit

- Turnaround time – amount of time to execute a particular process
- Waiting time – amount of time a process has been waiting in the ready queue
- Response time – amount of time it takes from when a request was submitted until the first response is produced.

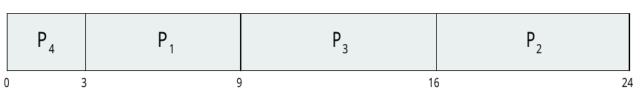
4.1 FCFS

FCFS = First-Come, First-Served Put the processes on the CPU the order in which they request it. (this is non-preemptive) Convoy effect - short process behind long process (this results in optimal average waiting time)
Example: For processes with burst times $P_1 = 24$; $P_2 = 3$; $P_3 = 3$; This is the resulting Gantt Chart:

If the processes come in different order, it is better.

4.2 SJF

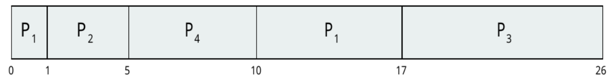
SJF = Shortest Job First Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time. Optimal average waiting time.
Non-Preemptive version: For processes with burst times $P_1 = 6$; $P_2 = 8$; $P_3 = 7$; $P_4 = 3$ This is the resulting Gantt Chart:



Determining the length of the next CPU-burst: ask the user to provide estimation / estimate ourselves.
Exponential averaging = Use length of previous CPU-burst and estimate the next one:
t_n = length on n-th CPU burst
τ_{n+1} = predicted next CPU burst
$\alpha, 0 \geq \alpha \leq 1$. Commonly $\alpha = \frac{1}{2}$
$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$

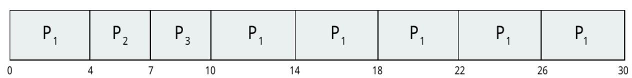
4.3 SRT

SRT = Shortest remaining time first
Preemptive version of SJF: For processes with arrival, burst times $P_1 = 0 \quad 8$; $P_2 = 1 \quad 4$; $P_3 = 2 \quad 9$; $P_4 = 3 \quad 5$ This is the resulting Gantt Chart:

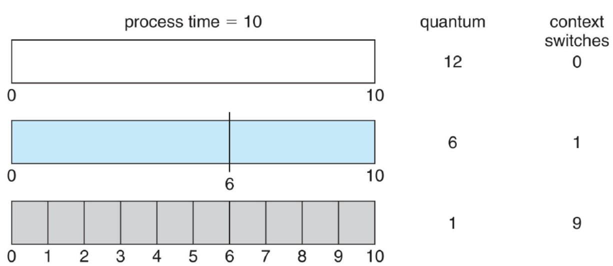


4.4 Round-Robin (RR)

Each process gets a small unit of CPU time (time quantum q). After this time has elapsed, the process is preempted and added to the end of the ready queue.
Preemptive RR: For processes with burst times: $P_1 = 24$; $P_2 = 3$; $P_3 = 3$ and q = 4, the Gantt chart is:

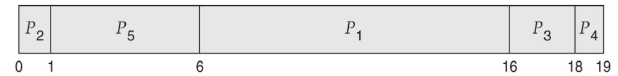


Choose q carefully, take into account context switches, and waiting time of processes:



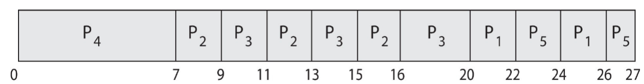
4.5 Priority Scheduling

A priority number (integer) is associated with each process. The CPU is allocated to the process with the highest priority (smallest integer = highest priority). SJF is priority scheduling.
Starvation = low priority processes may never execute. Can fix with **aging** = as time progresses increase the priority of the process.
Non Preemptive Priority Scheduling: For processes with burst times and priority: $P_1 = 10 \quad 3$; $P_2 = 1 \quad 1$; $P_3 = 2 \quad 4$; $P_4 = 1 \quad 5$; $P_5 = 5 \quad 2$, the Gantt chart is:



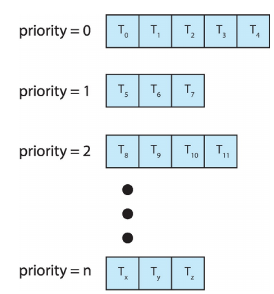
4.6 Priority scheduling with Round Robin

Run the process with the highest priority. Processes with the same priority run round-robin.
Example: For processes with burst and priority: $P_1 = 4 \quad 3$; $P_2 = 5 \quad 2$; $P_3 = 8 \quad 2$; $P_4 = 7 \quad 1$; $P_5 = 3 \quad 3$ and $q = 2$



4.7 Multilevel Queue

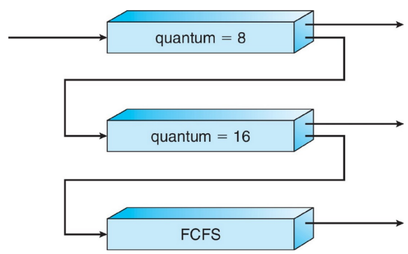
The ready queue consists of multiple queues.
Each queue has it's own scheduling algorithm + method to determine which queue a process goes to.
Example, multilevel queue by priority:



Multilevel Feedback Queue = A process can move between the various queues + method to determine which queue a process goes to + method to determine when to upgrade/demote process.

Example (Three queues):

- Q_0 = RR with q = 8ms
- Q_1 = RR with q = 16ms
- Q_2 = FCFS



When a process gains CPU, the process receives 8 ms. If it does not finish in 8 milliseconds, the process is moved to Q_1. If the process doesn't finish in 16 ms, it goes to FCFS.

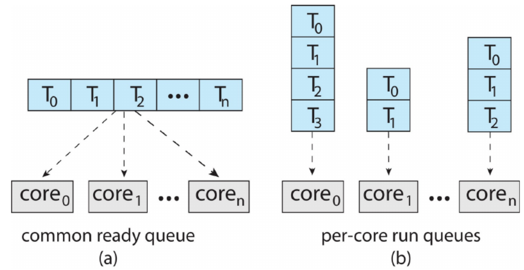
4.8 Thread scheduling

When threads are supported, threads are scheduled, not processes. Use the same algorithms.
Process-contention scope (PCS) = competition within the process. (on many-to-many / one-to-many threading models)
System-contention scope (SCS) = competition among all threads in system (on one-to-one models)

4.9 Scheduling on paralell systems

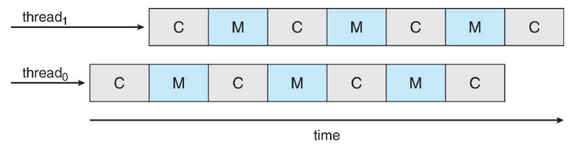
4.9.1 Multiple-Processor Scheduling

Symmetric multiprocessing (SMP) = each processor is self scheduling.
All threads may be in a common ready queue (a) or each processor may have its own private queue of threads (b).



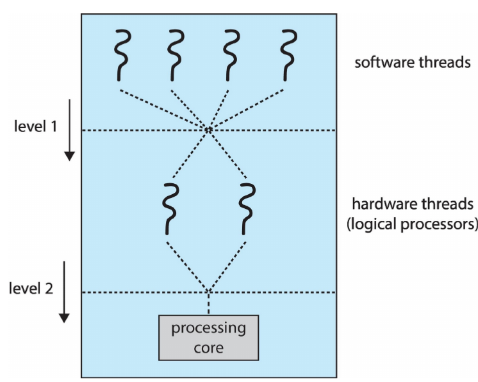
4.9.2 Multithreaded Mulitcore systems

Each core has 1 hardware threads. Then if one thread stalls, switch to another one. If they end up interleaved we have a speedup.



This is called Chip-multithread (hyperthreading) when you have multiple hardware threads on multiple cores. The systems sees each hw thread as a logical processor.
There are two levels of scheduling here:

- 1. The operating system deciding which software thread to run on a logical CPU
- 2. How each core decides which hardware thread to run on the physical core.



4.9.3 Load Balancing

: If the arhitecture is SMP, we need to keep all CPUs loaded for efficiency:

- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor

4.9.4 Processor Affinity

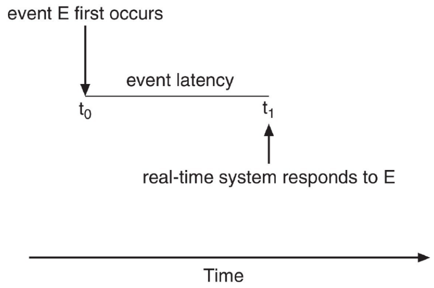
When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread. This is called **Affinity**.
Load balancing may affect affinity.
Solutions:

- Soft affinity – the operating system attempts to keep a thread running on the same processor, but no guarantees.
- Hard affinity – allows a process to specify a set of processors it may run on.

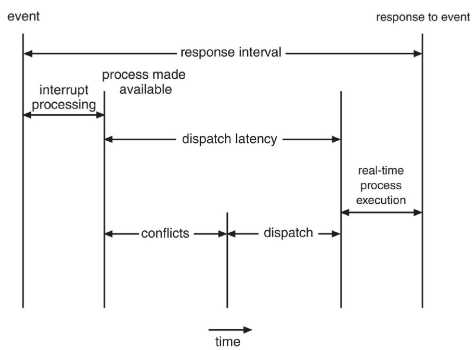
4.10 Real-time systems

Soft real-time systems = Critical real-time tasks have the highest priority, but no guarantee as to when tasks will be scheduled
Hard real-time systems = task must be serviced by its deadline
Event latency = the amount of time that elapses from when an event occurs to when it is serviced.
Two types of latencies affect performance:

- Interrupt latency = time from arrival of interrupt to start of routine that services interrupt.
- Dispatch latency – time for schedule to take current process off CPU and switch to another



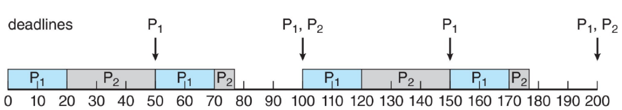
Interrupt latency consists of: 1) determining interrupt type and 2) switching the context.



Conflict phase of dispatch latency is: 1) preemption of any process running in kernel mode and 2) release by low-priority process of of resources needed by high-priority processes.

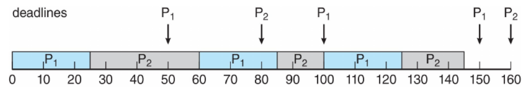
4.10.1 Real-time Priority-based Scheduling

For real-time scheduling, scheduler must support preemptive, priority- based scheduling.
Process may be periodic, have a processing time t, deadline d and period p. Rate of periodic task is $\frac{1}{p}$.
A priority is assigned based on the inverse of its period.



4.10.2 EDF

EDF = Earliest Deadline First Scheduling Priorities are assigned according to deadlines: The earlier the deadline, the higher the priority



4.10.3 Proportional Share Scheduling

T shares are allocated among all processes in the system
An application receives N shares where $N \leq T$.
This ensures each application will receive $\frac{N}{T}$ of the total processor time.

4.11 Little's formula

$n =$ average queue length $W =$ average waiting time in queue $\lambda =$ average arrival rate into queue $n = \lambda \cdot W$.