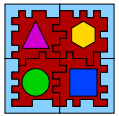


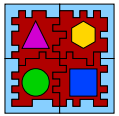
CPU Scheduling

Mateus Martínez de Lucena
Prof. Antônio Augusto Fröhlich, Ph.D.

UFSC / LISHA
June 15, 2022

Based on Silberschatz, Galvin and Gagne

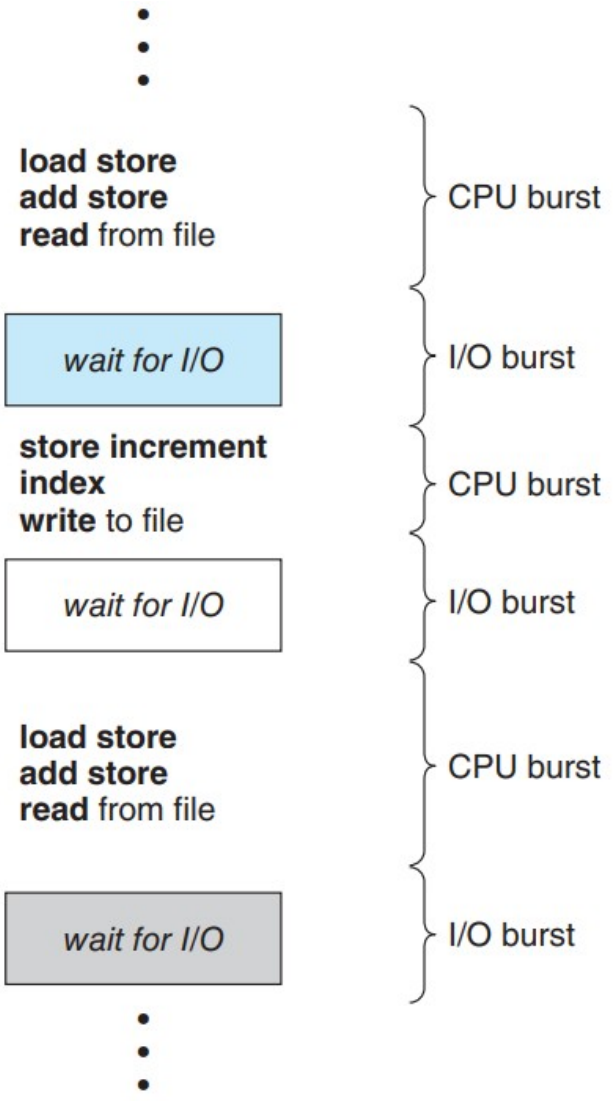
CPU Scheduling



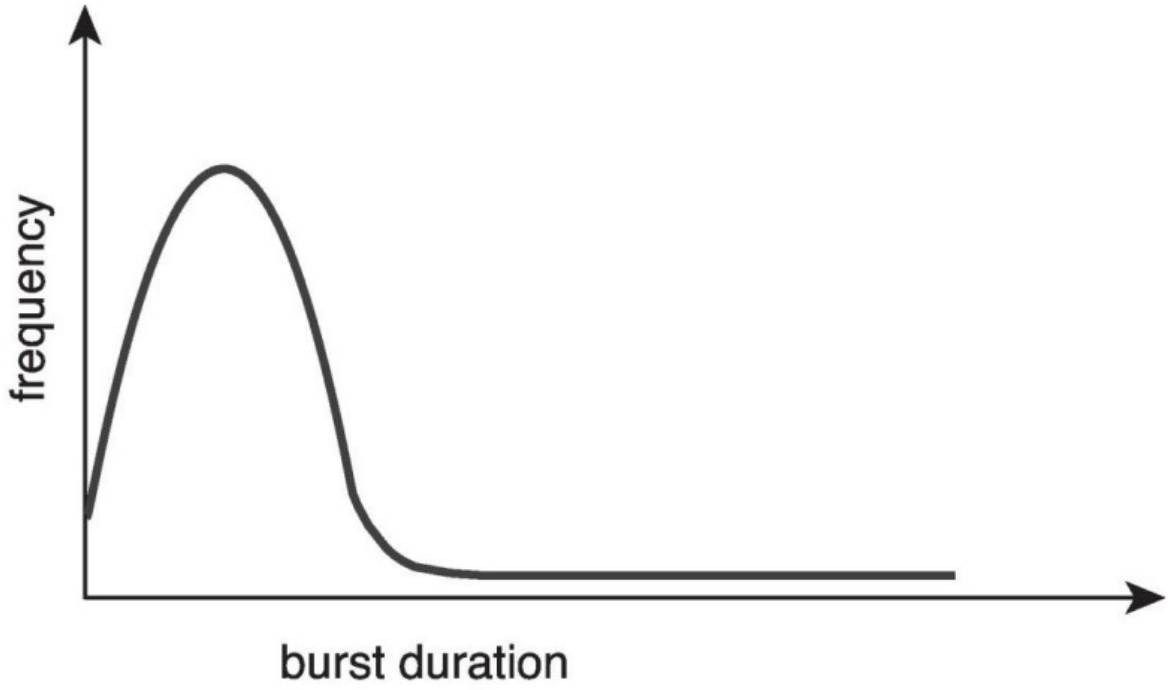
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multi-Processor Scheduling
- Operating Systems Examples
- Algorithm Evaluation

Basic Concepts

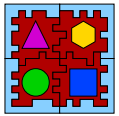
- Maximum CPU utilization
- CPU–I/O Burst Cycle
- CPU burst distribution is of main concern



Histogram of CPU-burst Times

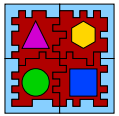


CPU Scheduler



- The **CPU scheduler** selects from among the processes in ready queue, and allocates a CPU core to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1) Switches from running to waiting state
  2) Switches from running to ready state
  3) Switches from waiting to ready
  4) Terminates
- Scheduling under 1 and 4 is **nonpreemptive**

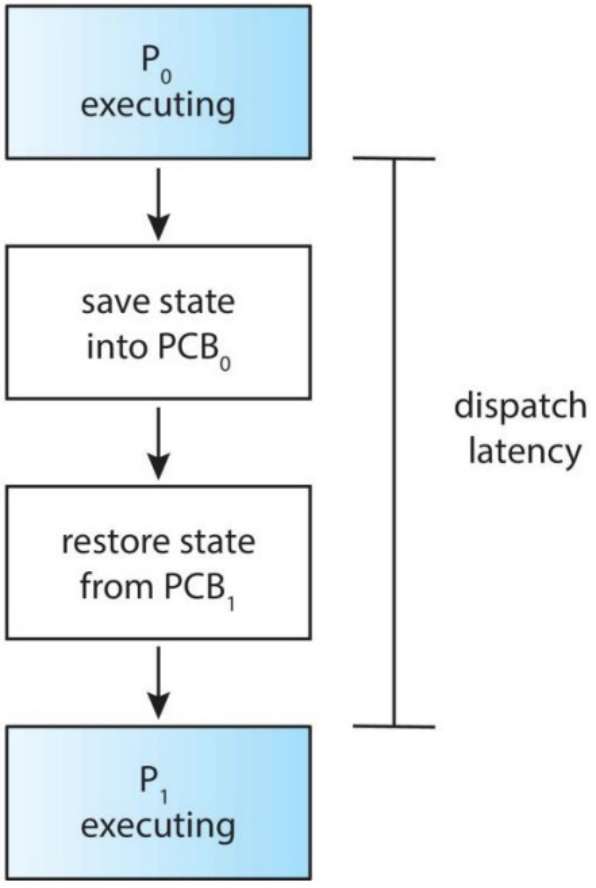
CPU Scheduler



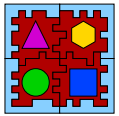
- All other scheduling is **preemptive**
  - Access to shared data
  - Preemption while in kernel mode
  - Interrupts occurring during crucial OS activities

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running



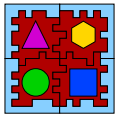
Dispatcher



- vmstat 1 3

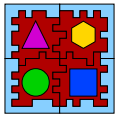
```
lucena@lucena-Latitude-7490: ~
File  Edit  View  Search  Terminal  Help
lucena@lucena-Latitude-7490:~$ vmstat 1 3
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 1  0      0 5073580 841216 6158172    0    0     2    30   14    8 11  4 85  0  0
 0  0      0 5073076 841216 6159052    0    0     0     0  440  793  1  1 98  0  0
 0  0      0 5073320 841216 6159060    0    0     0     0  489 1084  1  1 98  0  0
lucena@lucena-Latitude-7490:~$
```

Scheduling Criteria



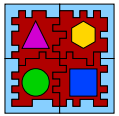
- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

Scheduling Algorithm Optimization Criteria



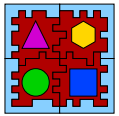
- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

Scheduling Algorithms



- First-Come, First-Served (FCFS)
- Shortest-Job-First (SJF)
- Round Robin (RR)
- Priority
- Multilevel Queue
- Multilevel Feedback Queue

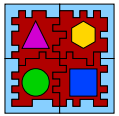
First-Come, First-Served (FCFS) Scheduling



Process	Burst Time	Order 1	Order 2
P1	24	P1	P2
P2	3	P2	P3
P3	3	P3	P1

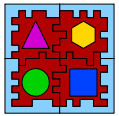
- Wait time?
- Order 2 < Order 1
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

Shortest-Job-First (SJF) Scheduling



- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user

Determining Length of Next CPU Burst

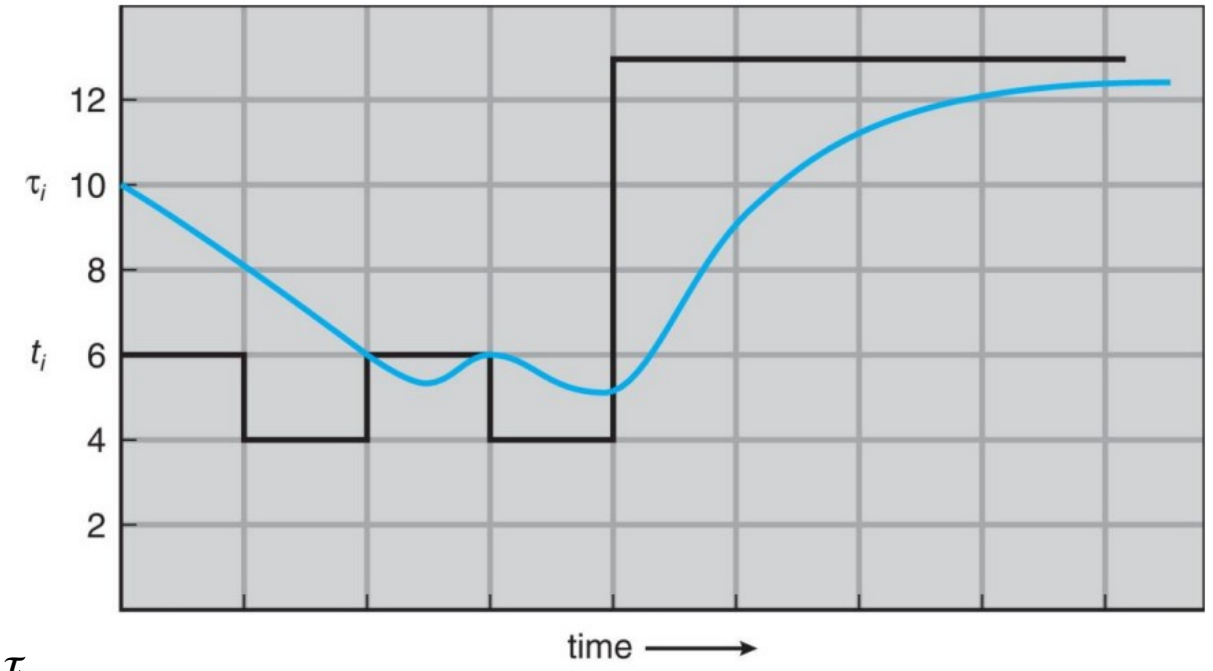


- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst

- Can be done by using the length of previous CPU bursts, using exponential averaging
  1) t_n = actual length of n^{th} CPU burst
  2) τ_{n+1} = predicted value of the next CPU burst
  3) α, $0 \leq \alpha \leq 1$
  4) Define: $\tau_{n+1} = \alpha t_n + (1-\alpha)\,\tau_n$

- Commonly, α set to ½
- Preemptive version called **shortest-remaining-time-first**

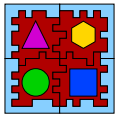
Prediction of the Length of the Next CPU Burst



$$\tau_{n+1} = \alpha t_n + (1-\alpha)\,\tau_n$$

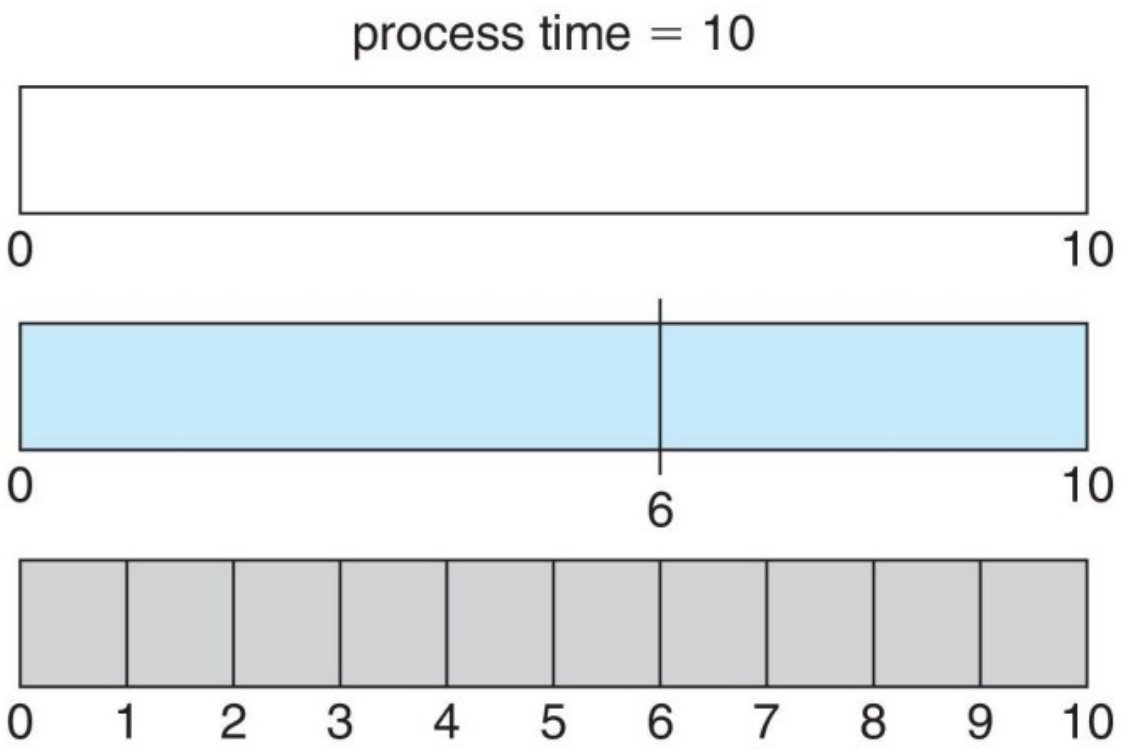
CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Round Robin (RR)



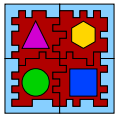
- Each process gets a small unit of CPU time (**time quantum** *q*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets *1/n* of the CPU time in chunks of at most q time units at once. No process waits more than *(n-1)q* time units.
- Timer interrupts every quantum to schedule next process
- Performance
  - q large → FIFO
  - q small → q must be large with respect to context switch, otherwise overhead is too high

Time Quantum and Context Switch Time

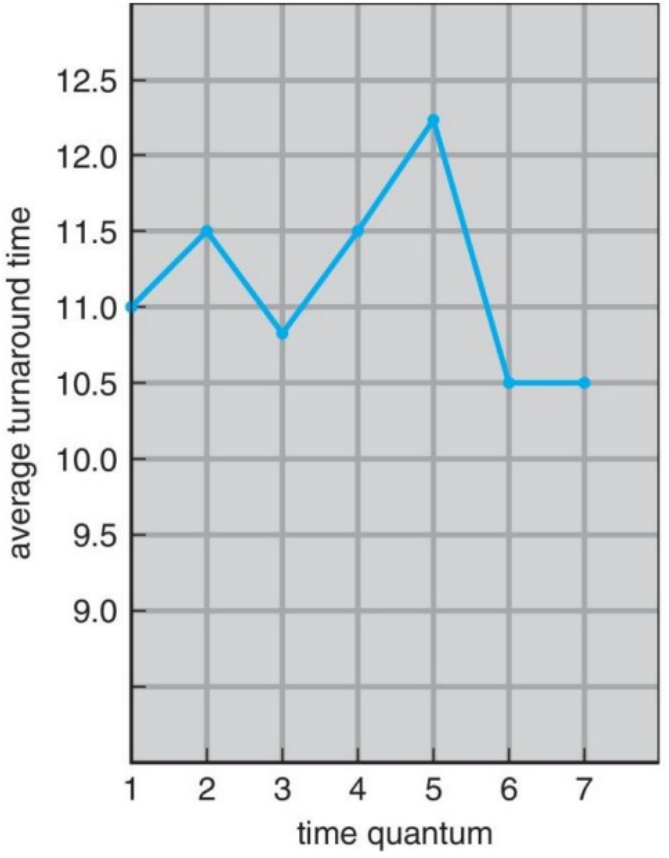


quantum	context switches
12	0
6	1
1	9

Turnaround Time Varies With The Time Quantum

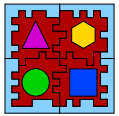


- 80% of CPU bursts should be shorter than *q*



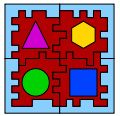
process	time
P_1	6
P_2	3
P_3	1
P_4	7

Priority Scheduling



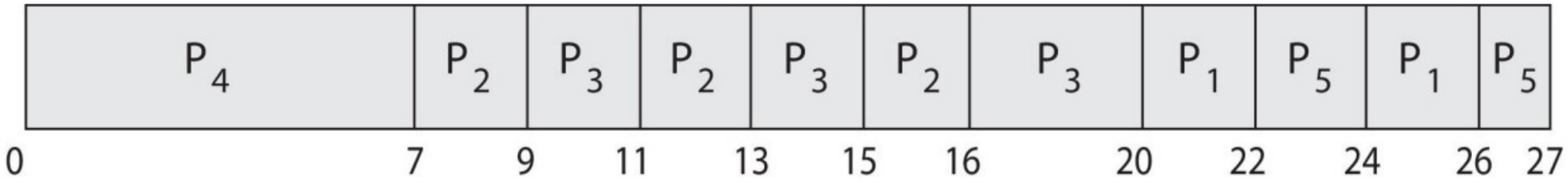
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer = highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem = **Starvation** – low priority processes may never execute
- Solution = **Aging** – as time progresses increase the priority of the process

Priority Scheduling w/ Round-Robin



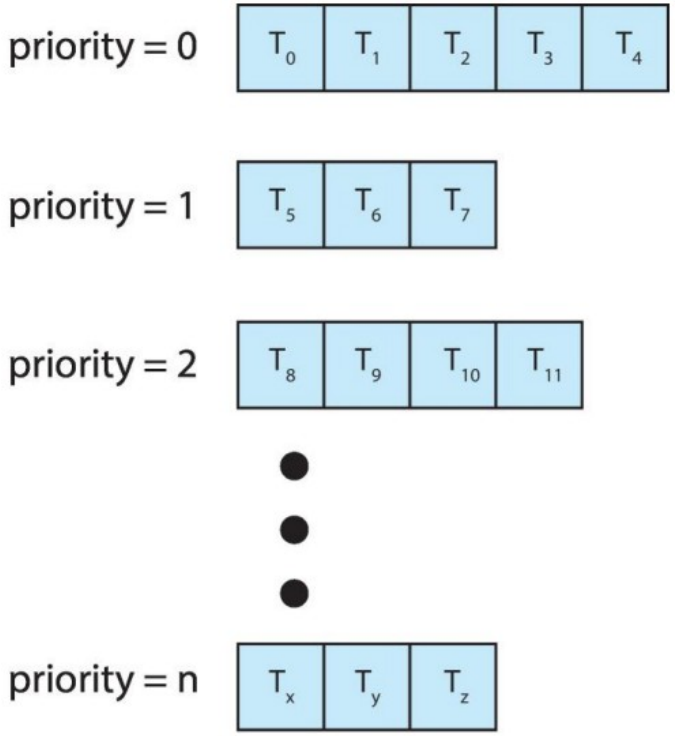
Process	Burst Time	Priority
P1	4	3
P2	5	2
P3	8	2
P4	7	1
P5	3	3

- Run the process with the highest priority. Processes with the same priority run round-robin
- Gantt Chart with 2 ms time quantum



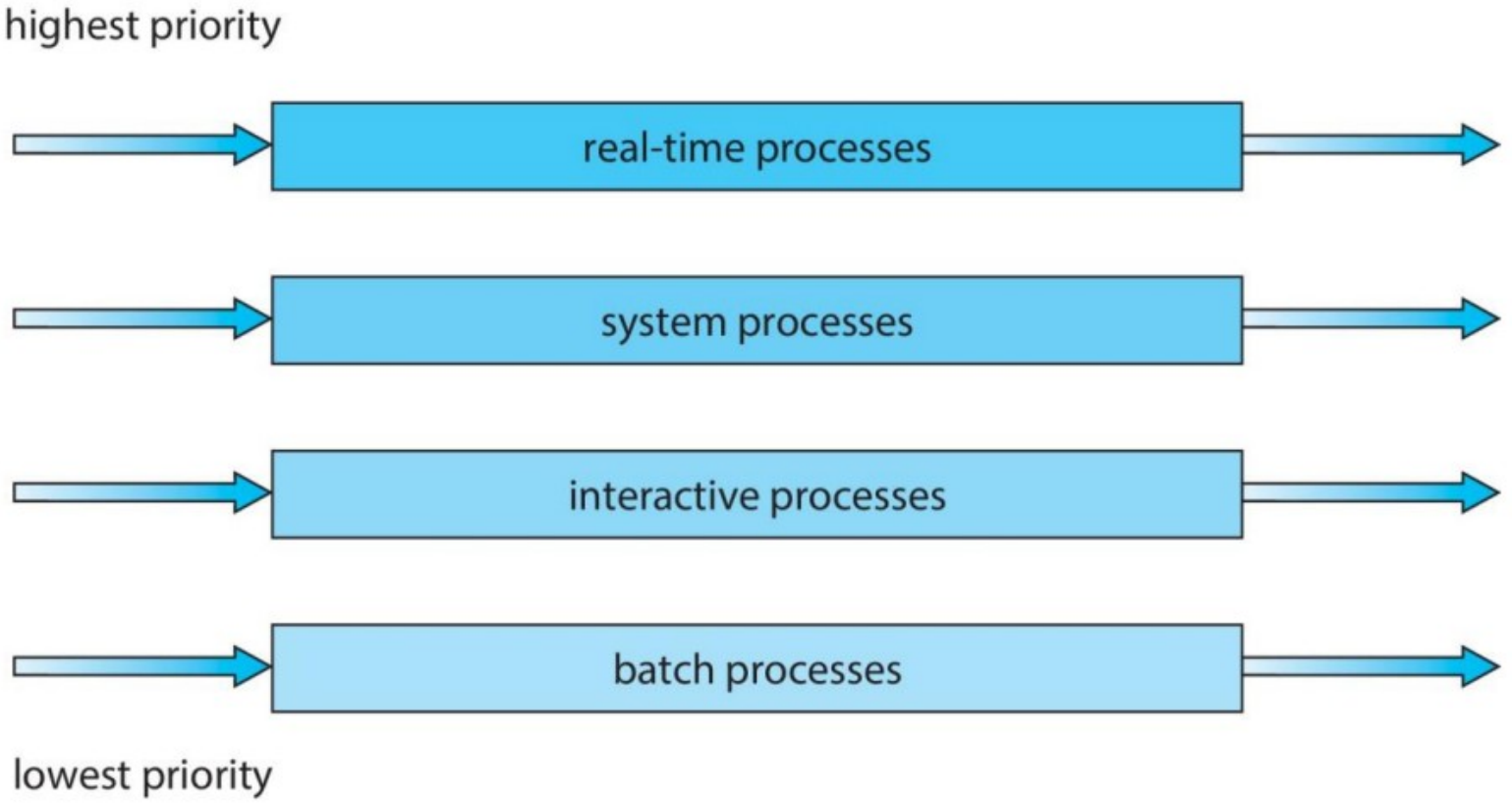
Multilevel Queue

- With priority scheduling, have separate queues for each priority.
- Schedule the process in the highest-priority queue!

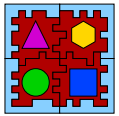


Multilevel Queue

- Prioritization based upon process type

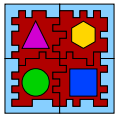


Multilevel Feedback Queue

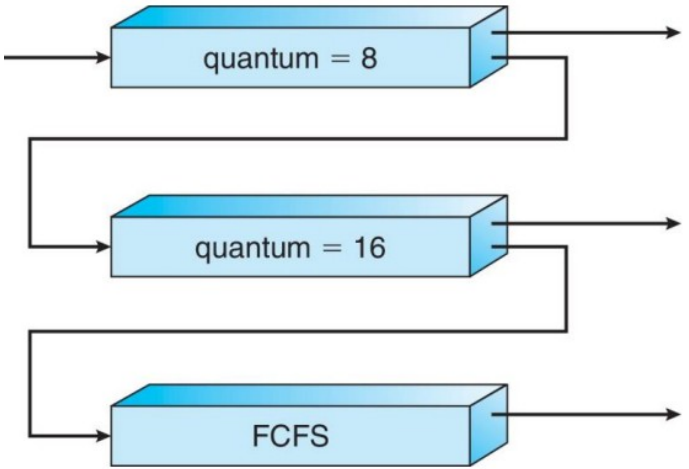


- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

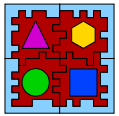
Example of Multilevel Feedback Queue



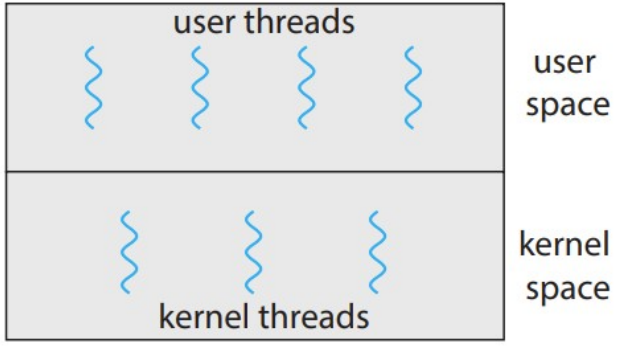
- Three queues:
  - Q0 – RR with time quantum 8 milliseconds
  - Q1 – RR time quantum 16 milliseconds
  - Q2 – FCFS
- Scheduling
  - A new job enters queue Q0 which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q1
  - At Q1 job is again served FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q2



Thread Scheduling

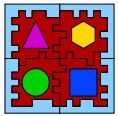


- Distinction between user-level and kernel-level threads

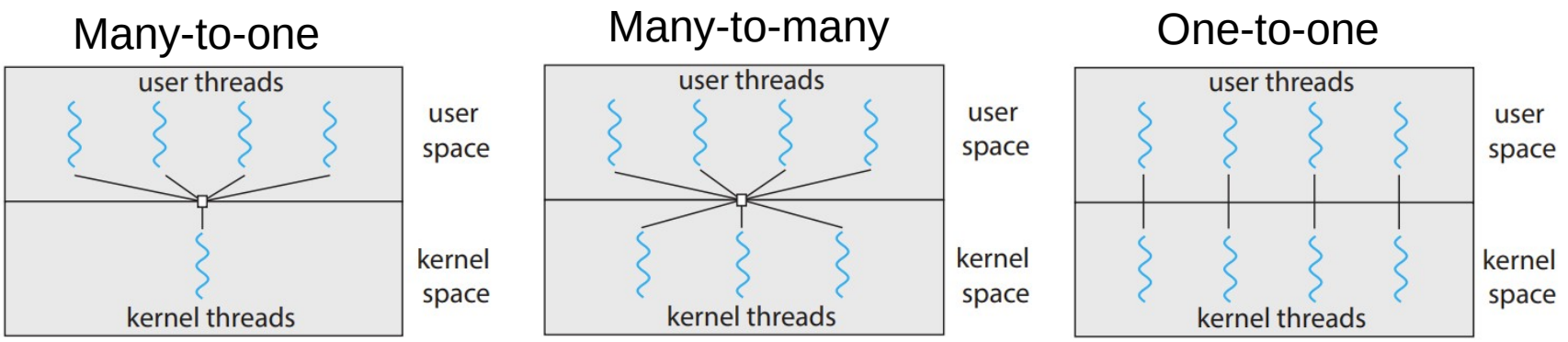


- When threads supported, threads scheduled, not processes

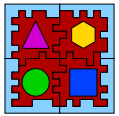
Thread Scheduling



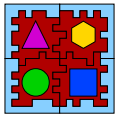
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on **Light Weight Processes** (LWP)
  - Known as **process-contention scope** (PCS) since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope** (SCS) – competition among all threads in system



Pthread Scheduling



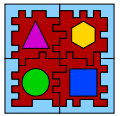
- API allows specifying either PCS or SCS during thread creation
  - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
  - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and macOS only allow PTHREAD_SCOPE_SYSTEM



```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[]) {
   int i, scope;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;
/* get the default attributes */
   pthread_attr_init(&attr);
/* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
   fprintf(stderr, "Unable to get scheduling scope\
n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("main: PTHREAD_SCOPE_PROCESS\n");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
         printf("main: PTHREAD_SCOPE_SYSTEM\n");
      else
         fprintf(stderr, "Illegal scope value.\n");
}
```

```c
/* set the scheduling algorithm to PCS or SCS */
   pthread_attr_setscope(&attr,
PTHREAD_SCOPE_SYSTEM);
/* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_create(&tid[i],&attr,runner,NULL);
/* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function
*/
void *runner(void *param)
{
/* do some work ... */
   pthread_exit(0);
}
```



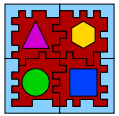
```
#include <pth
#include <std
#define NUM_T

int main(int
   int i, sc
   pthread_t
   pthread_at
/* get the de
   pthread_attr_init(&attr);
/* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
   fprintf(stderr, "Unable to get scheduling scope\
n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("main: PTHREAD_SCOPE_PROCESS\n");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
         printf("main: PTHREAD_SCOPE_SYSTEM\n");
      else
         fprintf(stderr, "Illegal scope value.\n");
}
```

```
lucena@lucena-Latitude-7490: ~/ws/so2
File  Edit  View  Search  Terminal  Help
lucena@lucena-Latitude-7490:~/ws/so2$ ./pthreadtest
main: PTHREAD_SCOPE_SYSTEM
PTHREAD_SCOPE_SYSTEM
PTHREAD_SCOPE_SYSTEM
PTHREAD_SCOPE_SYSTEM
PTHREAD_SCOPE_SYSTEM
PTHREAD_SCOPE_SYSTEM
lucena@lucena-Latitude-7490:~/ws/so2$
```

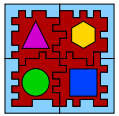
```
/* set the scheduling algorithm to PCS or SCS */
   pthread_attr_setscope(&attr,
PTHREAD_SCOPE_PROCESS);
/* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_create(&tid[i],&attr,runner,NULL);
/* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function
*/
void *runner(void *param)
{
/* do some work ... */
   pthread_exit(0);
}
```

Multiple-Processor Scheduling

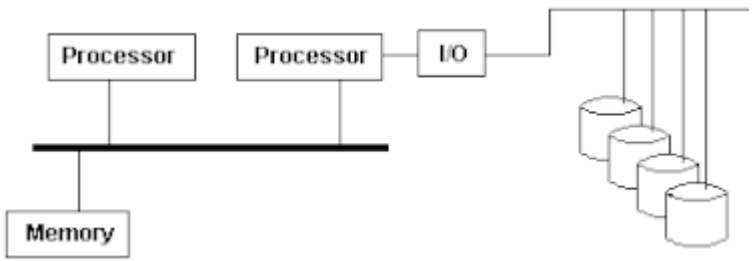


- CPU scheduling more complex when multiple CPUs are available
- Multiprocess may be any one of the following architectures:
  - Multicore CPUs
  - Multithreaded cores
  - NUMA systems
  - Heterogeneous multiprocessing

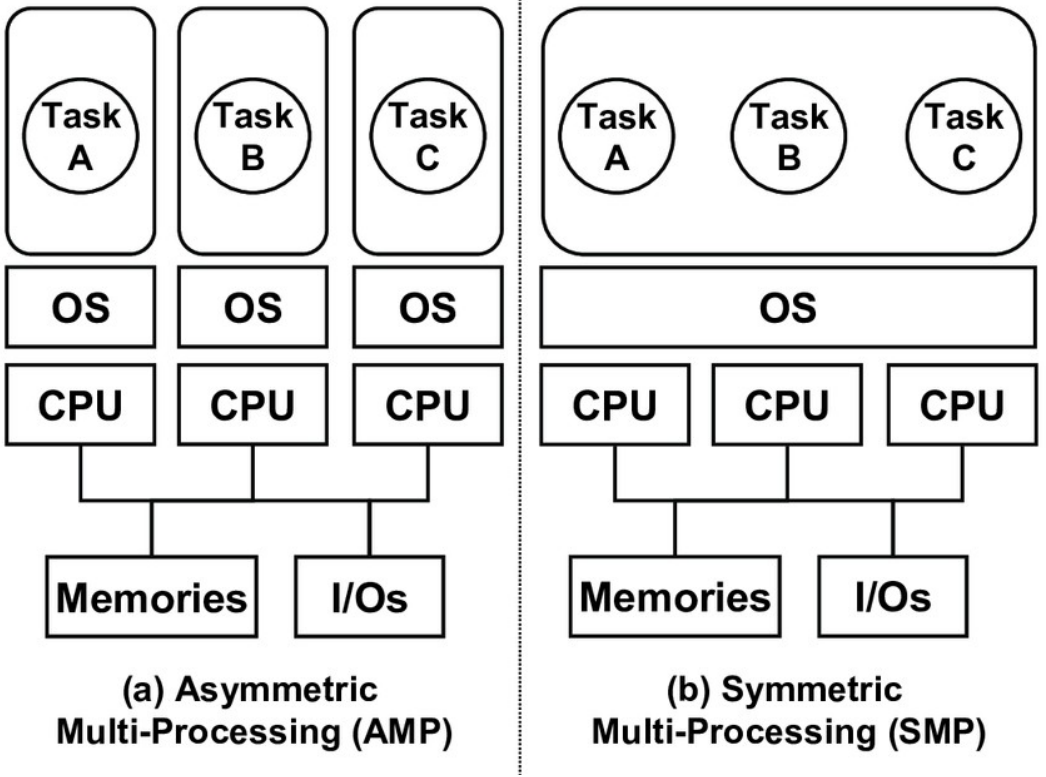
Symmetric vs Asymmetric Multiprocessing



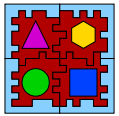
- Asymmetric multiprocessing
  - Single core accesses system data structures
  - Reduced data sharing
  - Main server becomes potential bottleneck



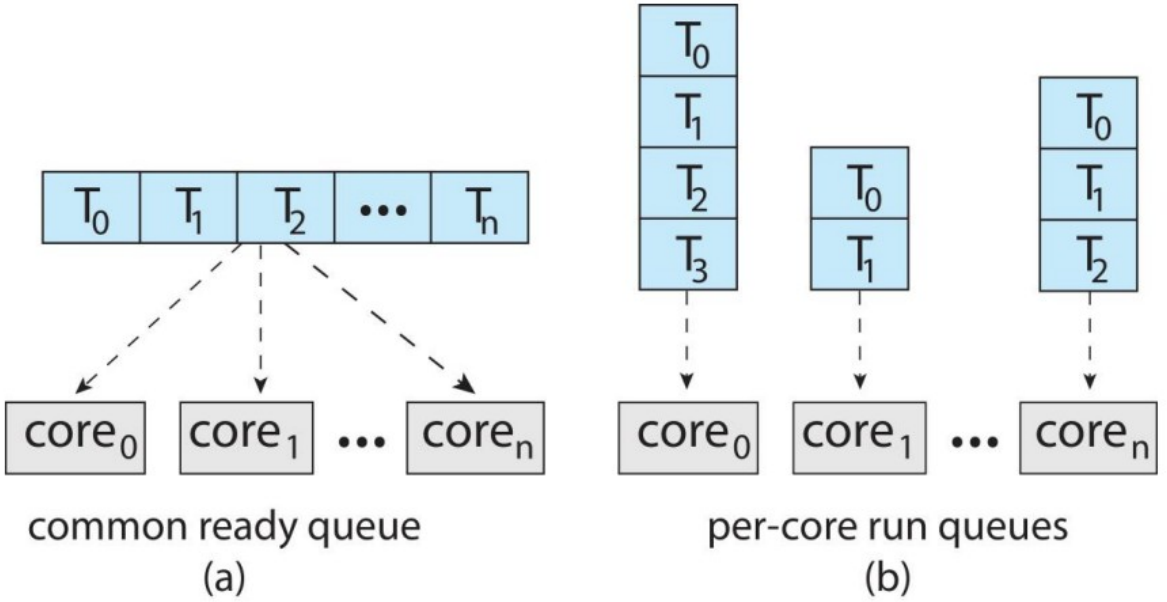
- Symmetric multiprocessing (SMP) is where each processor is self scheduling.



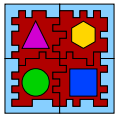
Multiple-Processor Scheduling



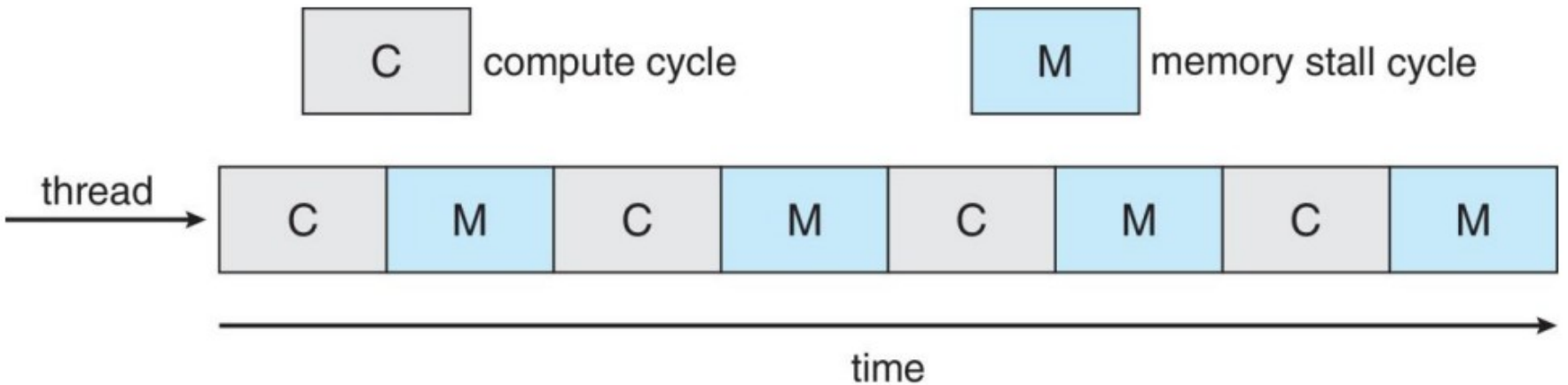
- All threads may be in a common ready queue (a)
  - Race condition on shared ready queue
- Each processor may have its own private queue of threads (b)



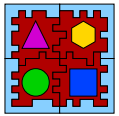
Multicore Processors



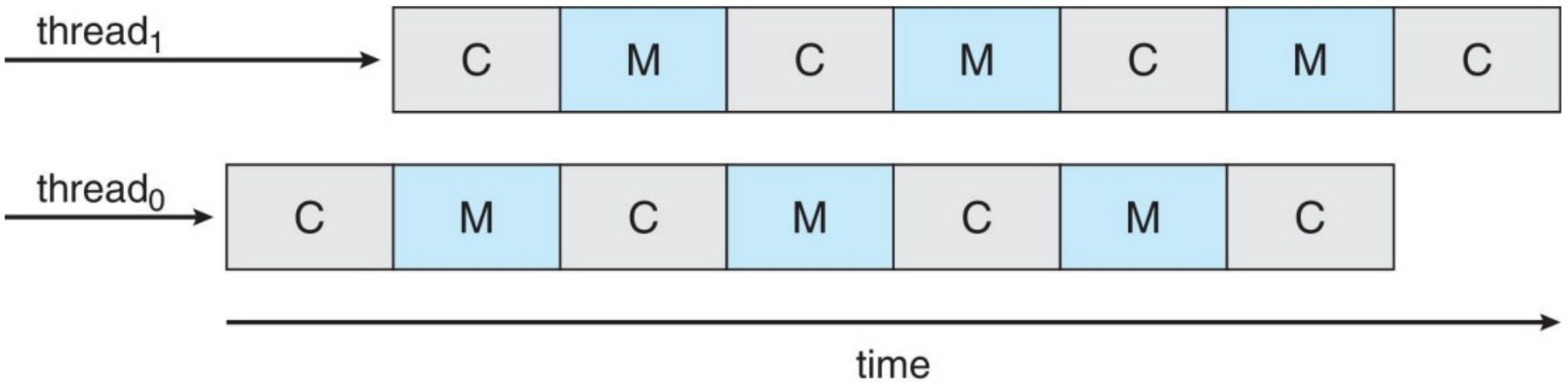
- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens



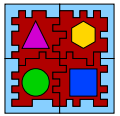
Multithreaded Multicore System



- Cores have multiple hardware threads
- One thread stalls, CPU switches to the other

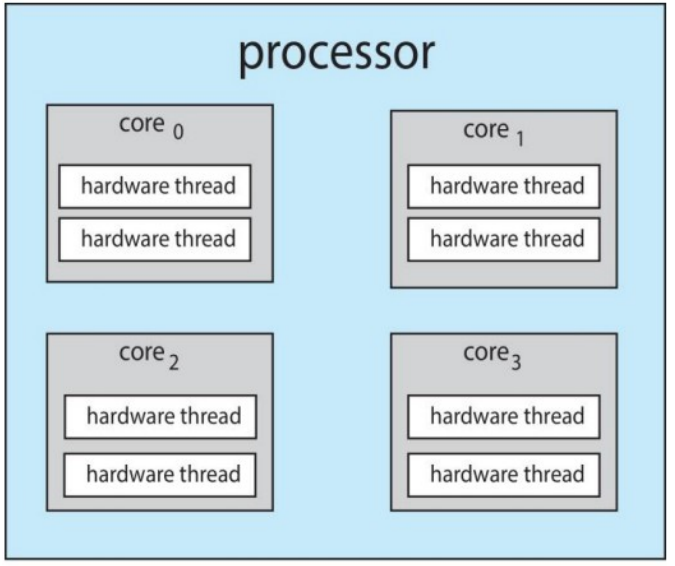


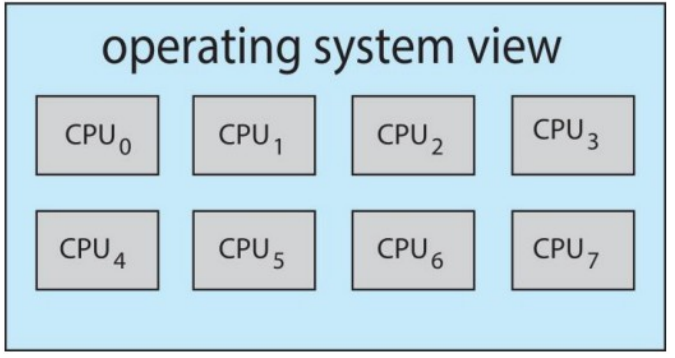
Multithreaded Multicore System



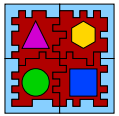
- Chip-multithreading (CMT) assigns each core multiple hardware threads. (Intel refers to this as hyperthreading.)

- On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors.





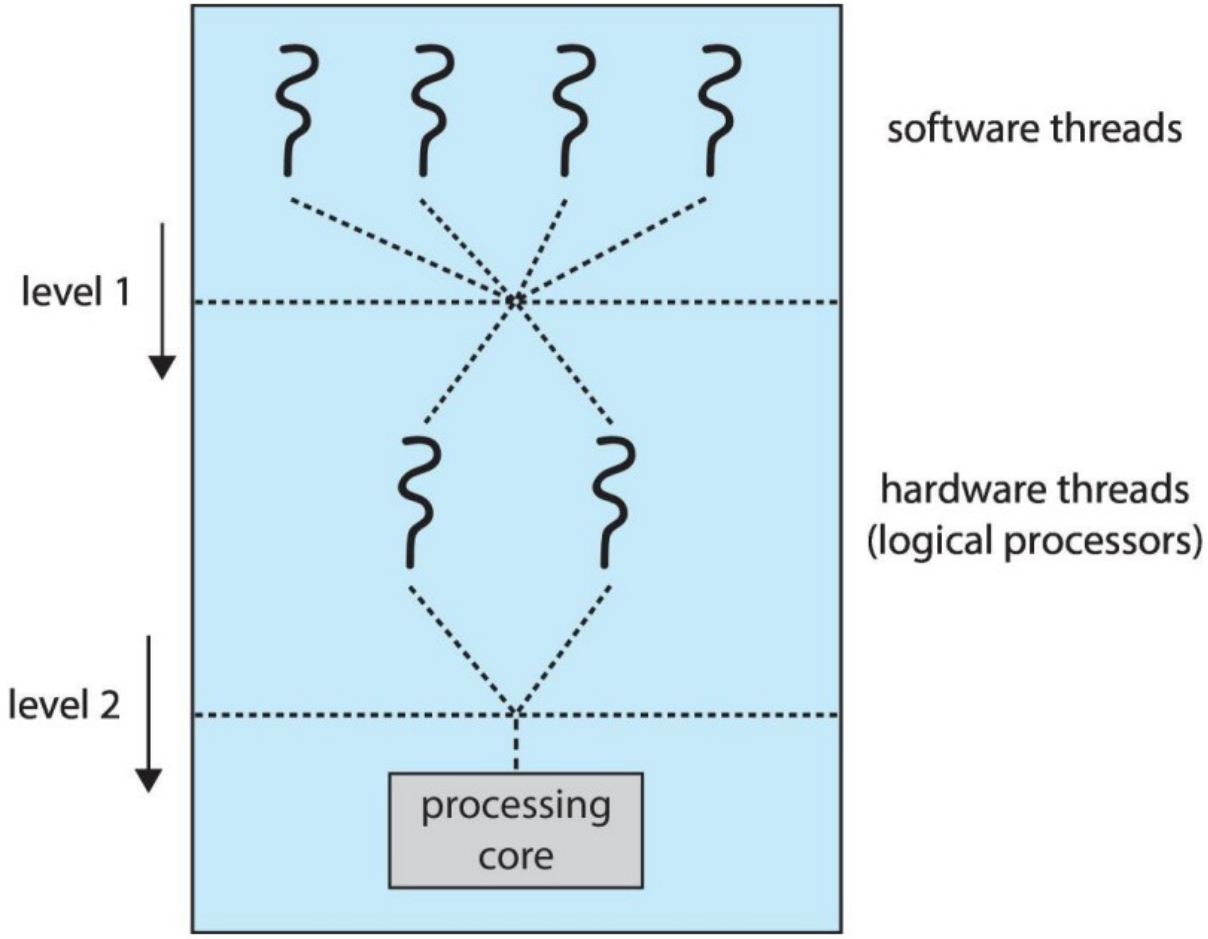
Multithreaded Multicore System



- Resources are shared:
  - Pipelines
  - Caches
- Multithreading
  - Coarse-grained
    - Thread executes until long latency event
    - Switching threads is expensive
      - Instruction pipeline must be flushed
  - Fine-grained (or interleaved)
    - Switches can happen at boundary of instruction cycle
    - Includes logic for thread switching
      - Low cost switching

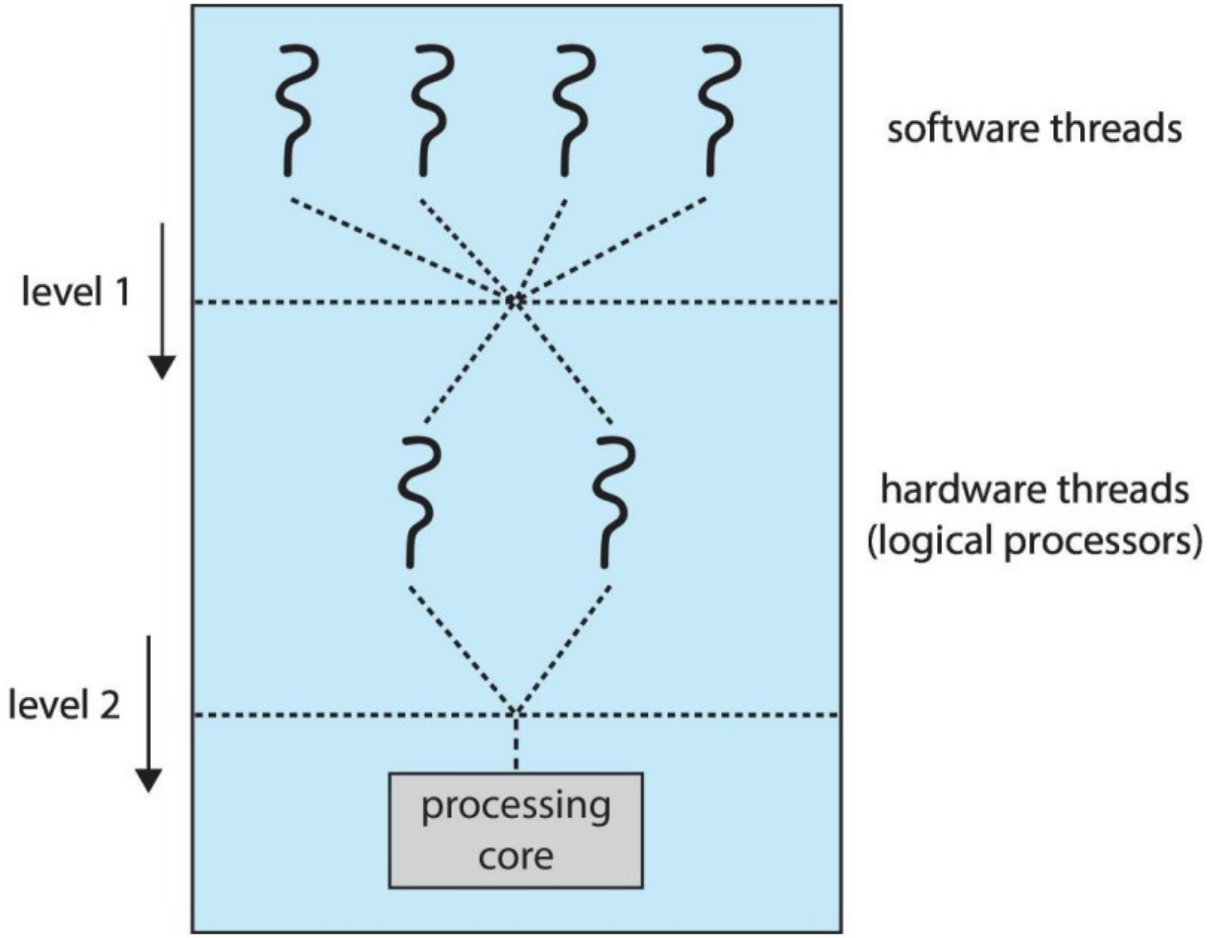
Multithreaded Multicore System

- Two levels of scheduling:
- 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.

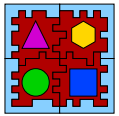


Multithreaded Multicore System

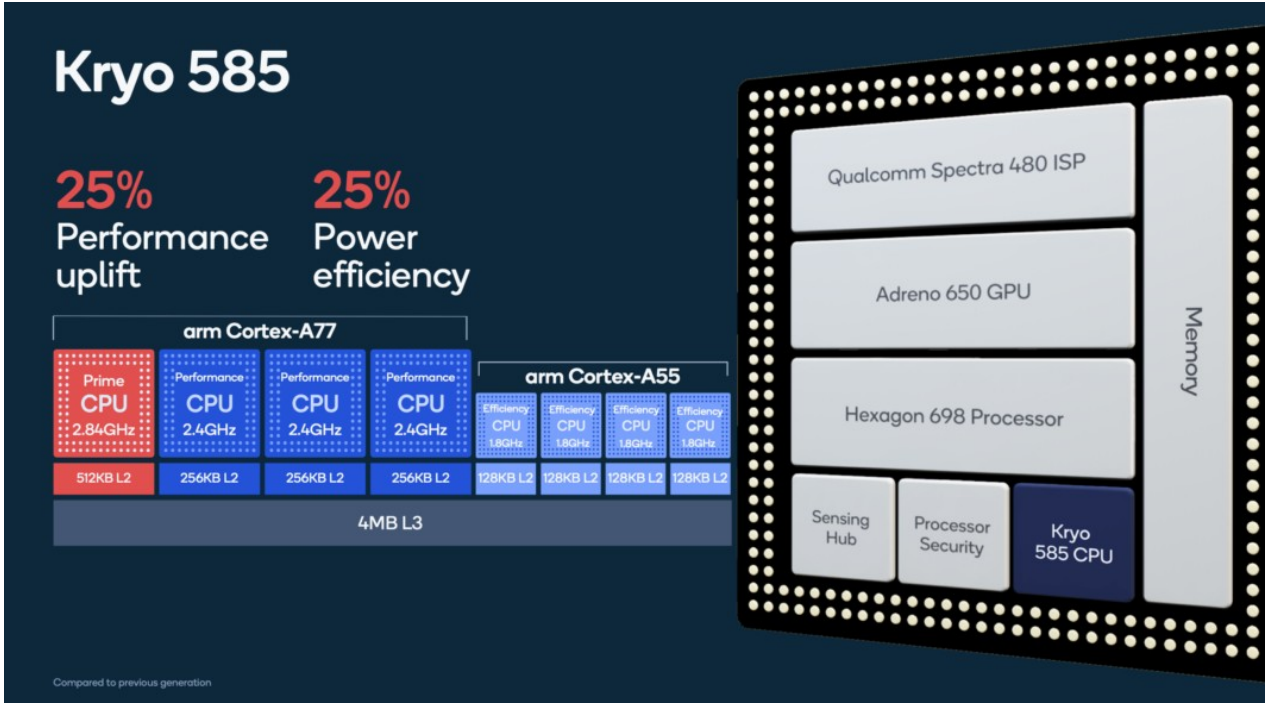
- UltraSPARC T3
  - Simple RR
- Intel Itanium
  - dynamic urgency 0 to 7
  - 0 – lowest
  - 7 – highest



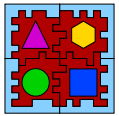
Heterogeneous Multiprocessing (HMP)



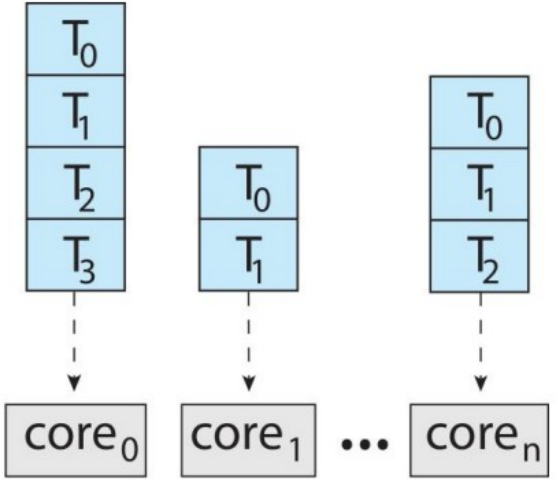
- Multiple cores
- Different clock speed/power management
- Manage power consumption
- Example: Snapdragon 865
  - 4 Cortex-A55 1,8 GHz
    - Simpler tasks
  - 3 Cortex-A77 2,42 GHz
    - More demanding apps
  - 1 Cortex-A77 2,84 GHz
    - Rarely used
    - For extreme tasks
    - Overheats the system



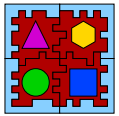
Multiple-Processor Scheduling – Load Balancing



- If SMP, 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
- Balancing metric
  - Even threads
  - Even priorities

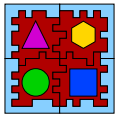


Multiple-Processor Scheduling – Processor Affinity



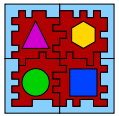
- When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread.
- We refer to this as a thread having affinity for a processor (i.e. "**processor affinity**")
- Load balancing may affect processor affinity as a thread may be moved from one processor to another to balance loads, yet that thread loses the contents of what it had in the cache of the processor it was moved off of.

Multiple-Processor Scheduling – Processor Affinity

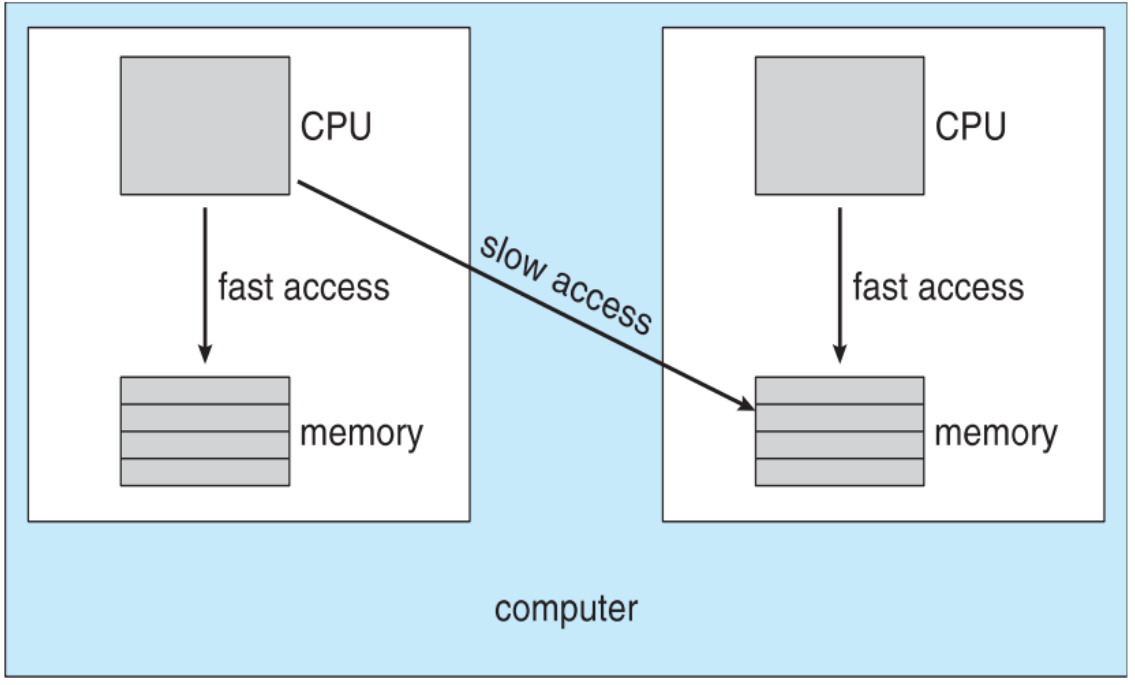


- **Soft affinity** – the operating system attempts to keep a thread running on the same processor, but no guarantees.
- **Hard affinity** – allows a process to specify a set of processors it may run on.

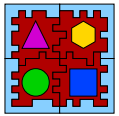
NUMA and CPU Scheduling



- If the operating system is **NUMA-aware**, it will assign memory closest to the CPU the thread is running on.

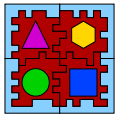


Operating System Examples



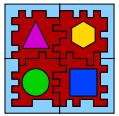
- Linux O(1) scheduler
- Linux CFS
- Windows scheduling
- Solaris scheduling
- MAC OS scheduling

Linux O(1) Scheduler



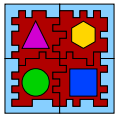
- Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm
- Version 2.5 moved to constant order O(1) scheduling time
  - Preemptive, priority based
  - 1 **runqeue** per CPU
  - Two priority ranges: time-sharing and real-time
  - Real-time range from 0 to 99 and nice value from 100 to 140
  - Map into global priority with numerically lower values indicating higher priority

Linux O(1) Scheduler



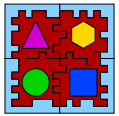
- Each CPU:
  - 1 runqeue
  - 2 priority arrays
    - 140 double linked lists
    - Active processes ← Tasks start here
    - Expired processes ← Ran out of **timeslice**
- Nice value -20 to 19
  - CPU-I/O bound tasks penalized/rewarded
    - Maximum adjusted 5
    - sleep_avg determines CPU vs I/O bound

Linux Completely Fair Scheduler



- Completely Fair Scheduler (CFS), since version 2.6.23
- Scheduling classes
  - Each has specific priority
  - Scheduler picks highest priority task in highest scheduling class
  - Rather than quantum based on fixed time allotments, based on proportion of CPU time
  - 2 scheduling classes included, others can be added
    - default
    - real-time

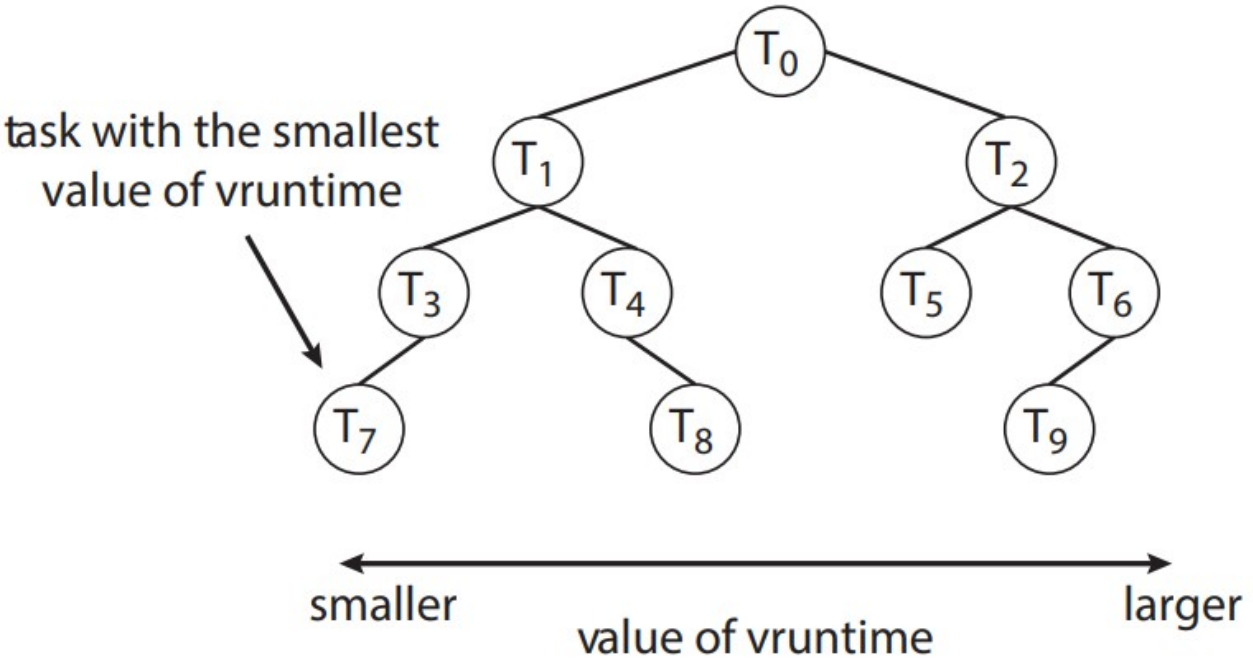
Linux Completely Fair Scheduler



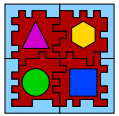
- Quantum calculated based on **nice value** from -20 to +19
  - Lower value is higher priority
  - Calculates **target latency** – interval of time during which task should run at least once
  - Target latency can increase if say number of active tasks increases
- CFS scheduler maintains per task **virtual run time** in variable *vruntime*
  - Associated with decay factor based on priority of task – lower priority is higher decay rate
  - Normal default priority yields virtual run time = actual run time
- To decide next task to run, scheduler picks task with lowest virtual run time

CFS Performance

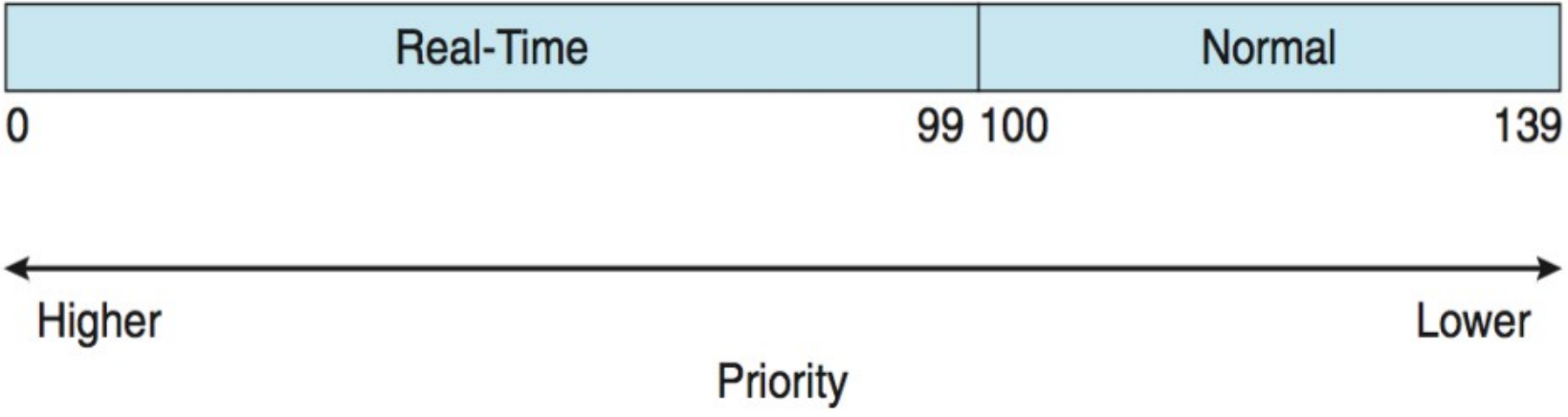
- Tasks placed in red-black tree
  - Key: *vruntime*
  - Only runnable tasks
  - O(log N) travel time
  - *rb_leftmost*



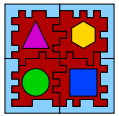
Linux Scheduling (Cont.)



- Real-time scheduling according to POSIX.1b
  - Real-time tasks have static priorities
- Real-time plus normal map into global priority scheme
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139

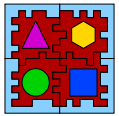


Windows Scheduling



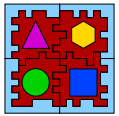
- Windows uses priority-based preemptive scheduling
- Highest-priority thread runs next
- **Dispatcher** is scheduler
- Thread runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
- Real-time threads can preempt non-real-time
- 32-level priority scheme
- **Variable class** is 1-15, **real-time class** is 16-31
- Priority 0 is memory-management thread
- Queue for each priority
- If no run-able thread, runs **idle thread**

Windows Priority Classes



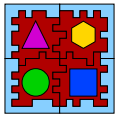
- Win32 API identifies several priority classes to which a process can belong
  - REALTIME_PRIORITY_CLASS, HIGH_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS,NORMAL_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, IDLE_PRIORITY_CLASS
  - All are variable except REALTIME
- A thread within a given priority class has a relative priority
  - TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE

Windows Priority Classes



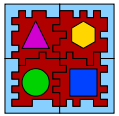
- Priority class and relative priority combine to give numeric priority
- Base priority is NORMAL within the class
- If quantum expires, priority lowered, but never below base
- If wait occurs, priority boosted depending on what was waited for
- Foreground window given 3x priority boost
- Windows 7 added **user-mode scheduling** (UMS)
  - Applications create and manage threads independent of kernel
  - For large number of threads, much more efficient
  - UMS schedulers come from programming language libraries like C++ **Concurrent Runtime** (ConcRT) framework

Windows Priorities



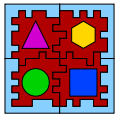
	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Solaris Scheduling



- Priority-based scheduling
- Six classes available
  - Time sharing (default) (TS)
  - Interactive (IA)
  - Real time (RT)
  - System (SYS)
  - Fair Share (FSS)
  - Fixed priority (FP)
- Given thread can be in one class at a time
- Each class has its own scheduling algorithm
- Time sharing is multi-level feedback queue
- Loadable table configurable by sysadmin

Solaris Scheduling: Dispatch Table

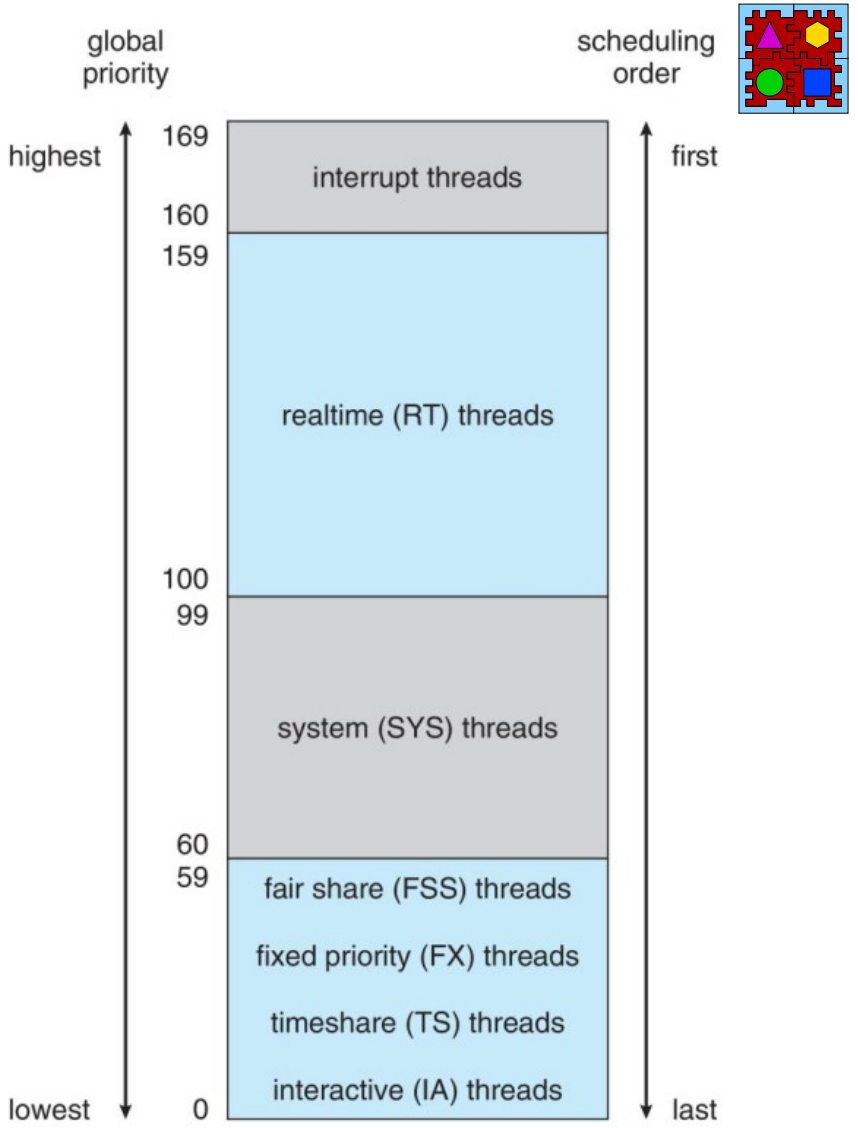


- Time-sharing and Interactive class
- Higher priority lower quantum
- Expired quantum lower priority
- Return from sleep higher priority

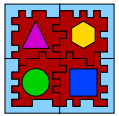
priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Solaris Scheduling

- Class priority maps into global priority
- Real-time always preempts kernel threads
- Fair-share and Fixed priority are not dynamic



Solaris Scheduling



- Scheduler converts class-specific priorities into a per-thread global priority
  - Thread with highest priority runs next
  - Runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
  - Multiple threads at same priority selected via RR

MAC OS Scheduling

- OS X Mach Scheduler
- OO scheduler
  - Task
    - Execution Environment
    - Basic unit of resource allocation
    - Protected access to system resources via ports
  - Thread
    - Basic unit of execution
    - Shared resources within task
  - Port
    - Kernel-protected communication channel
    - Port rights

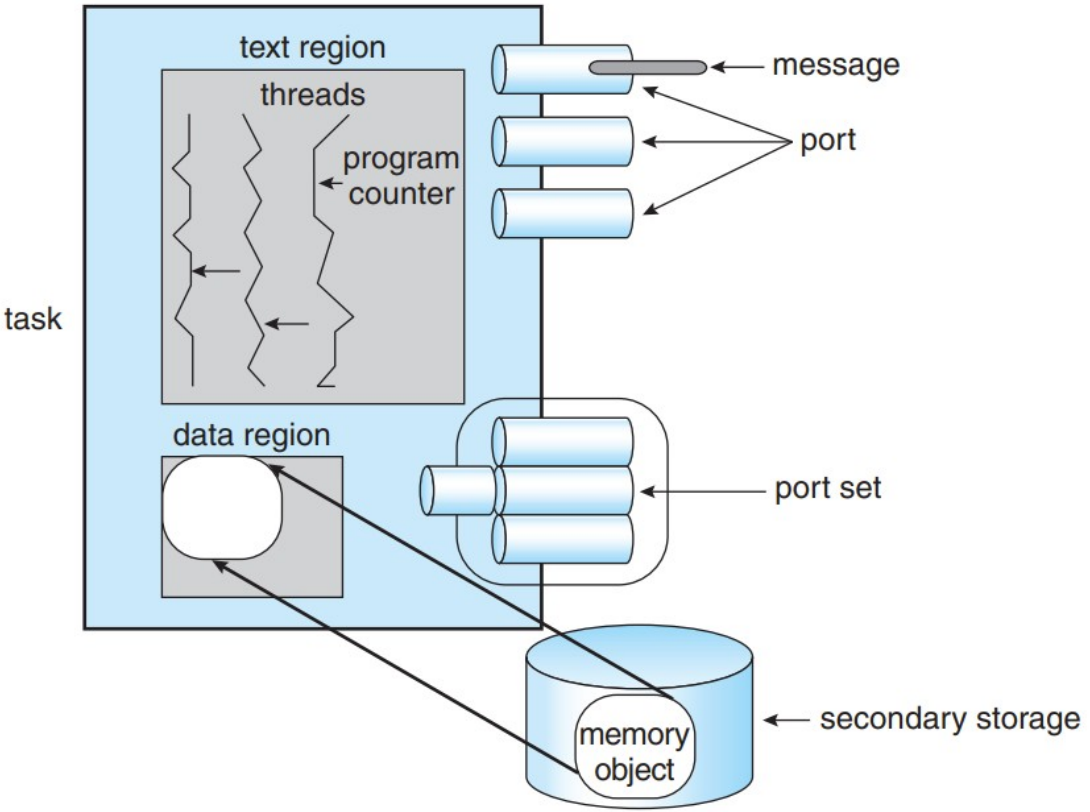


Figure B.2 Mach's basic abstractions.

MAC OS Scheduling

- OO scheduler
  - Port set
    - Common message queue
    - Messages identify port of arrival
  - Message
    - Typed collection of objects
    - Actual data or pointer
    - Port rights are passed through messages
  - Memory object
    - Tasks map into address space
    - Any object that can be memory mapped

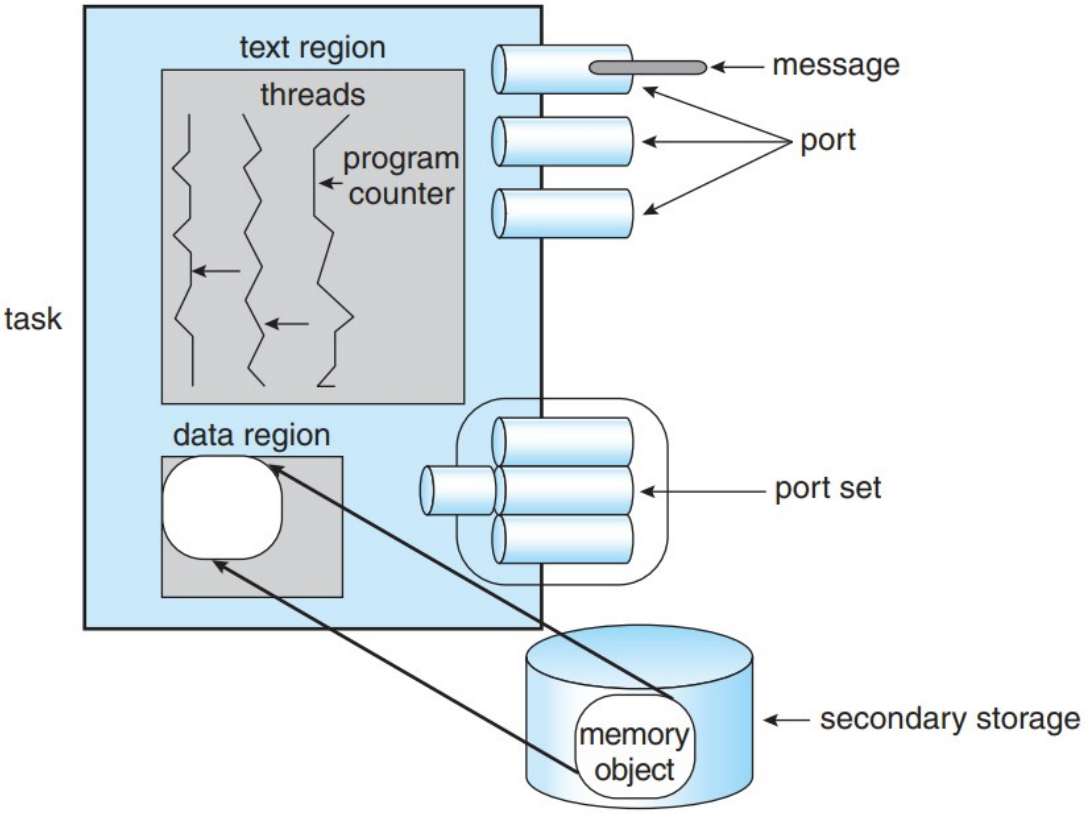
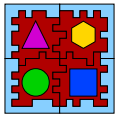


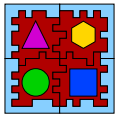
Figure B.2 Mach's basic abstractions.

MAC OS Scheduling



- No central dispatcher to assign threads to CPU
  - Each CPU consults local and global run queue to select threads
- Quantum is variable (inversely to total number of threads in system)
  - e.g. 10 processors, 11 threads, 100 ms quantum → context switch only once per second per processor

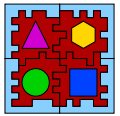
MAC OS Scheduling



- Priority based
  - Exponential average of CPU usage
  - Used to place threads in 32 global run queues
  - Local run queues for CPU specific threads

Range	Description
0-10	Lowest priorities (aged, idle)
11-30	Lowered priorities
31	Base for user threads
32-51	Elevated priorities
46	Background
47	Foreground
48	Control
52-63	Max priority
64-79	System reserved
80-95	Kernel Priorities
96-127	Real-time

MAC OS Scheduling

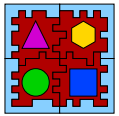


- Priority based
  - Exponential average of CPU usage
  - Used to place threads in 32 global run queues
  - Local run queues for CPU specific threads

Range	Description
0-10	Lowest priorities (aged, idle)
11-30	Lowered priorities
31	Base for user threads
32-51	Elevated priorities
46	Background

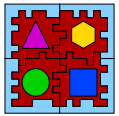
```
#define do_priority_computation(thread, pri) \
    do { \
        (pri) = (thread->priority) /* start with base priority */ \
        - ((thread)->sched_usage >> (thread)->pri_shift); \
        if ((pri) < MINPRI_USER) \
            (pri) = MINPRI_USER; \
        else if ((pri) > MAXPRI_KERNEL) \
            (pri) = MAXPRI_KERNEL; \
    } while (FALSE);
```

Algorithm Evaluation



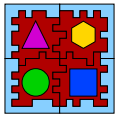
- How to select CPU-scheduling algorithm for an OS?
  - CPU utilization
  - Throughput
  - Turnaround time
  - Waiting time
  - Response time
- Determine criteria, then evaluate algorithms
  - Maximizing CPU utilization under the constraint that the maximum response time is 300 milliseconds
  - Maximizing throughput such that turnaround time is (on average) linearly proportional to total execution time

Algorithm Evaluation Methods



- Deterministic modeling
- Queueing Models
- Simulations
- Implementation

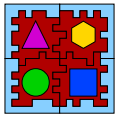
Algorithm Evaluation



- **Deterministic modeling**
  - Type of **analytic evaluation**
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Consider 5 processes arriving at time 0:

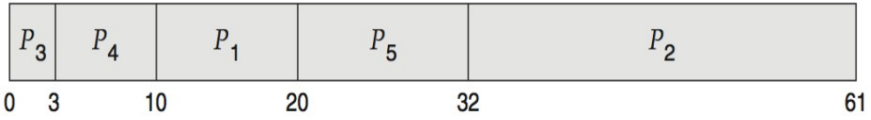
Process	Burst Time
P1	10
P2	29
P3	3
P4	7
P5	12

Deterministic Evaluation

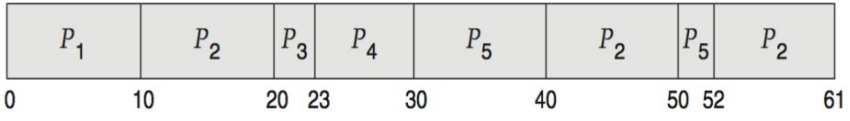


- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
  - FCS is 28ms:

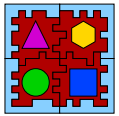
  - Non-preemptive SJF is 13ms:



  - RR is 23ms:

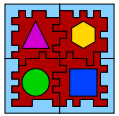


Queueing Models



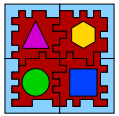
- Describes the arrival of processes, and CPU and I/O bursts probabilistically
  - Commonly exponential, and described by mean
  - Computes average throughput, utilization, waiting time, etc
- Computer system described as network of servers, each with queue of waiting processes
  - Knowing arrival rates and service rates
  - Computes utilization, average queue length, average wait time, etc

Little's Formula



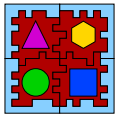
- n = average queue length
- W = average waiting time in queue
- λ = average arrival rate into queue
- Little's law – in steady state, processes leaving queue must equal processes arriving, thus:
  - n = λ x W
  - Valid for any scheduling algorithm and arrival distribution
- For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds

Simulations



- Queueing models limited
- Simulations more accurate
  - Programmed model of computer system
  - Clock is a variable
  - Gather statistics indicating algorithm performance
  - Data to drive simulation gathered via
    - Random number generator according to probabilities
    - Distributions defined mathematically or empirically
    - Trace tapes record sequences of real events in real systems

Implementation



- Even simulations have limited accuracy
- Just implement new scheduler and test in real systems
  - High cost, high risk
  - Environments vary
- Most flexible schedulers can be modified per-site or per-system
- Or APIs to modify priorities
- But again environments vary