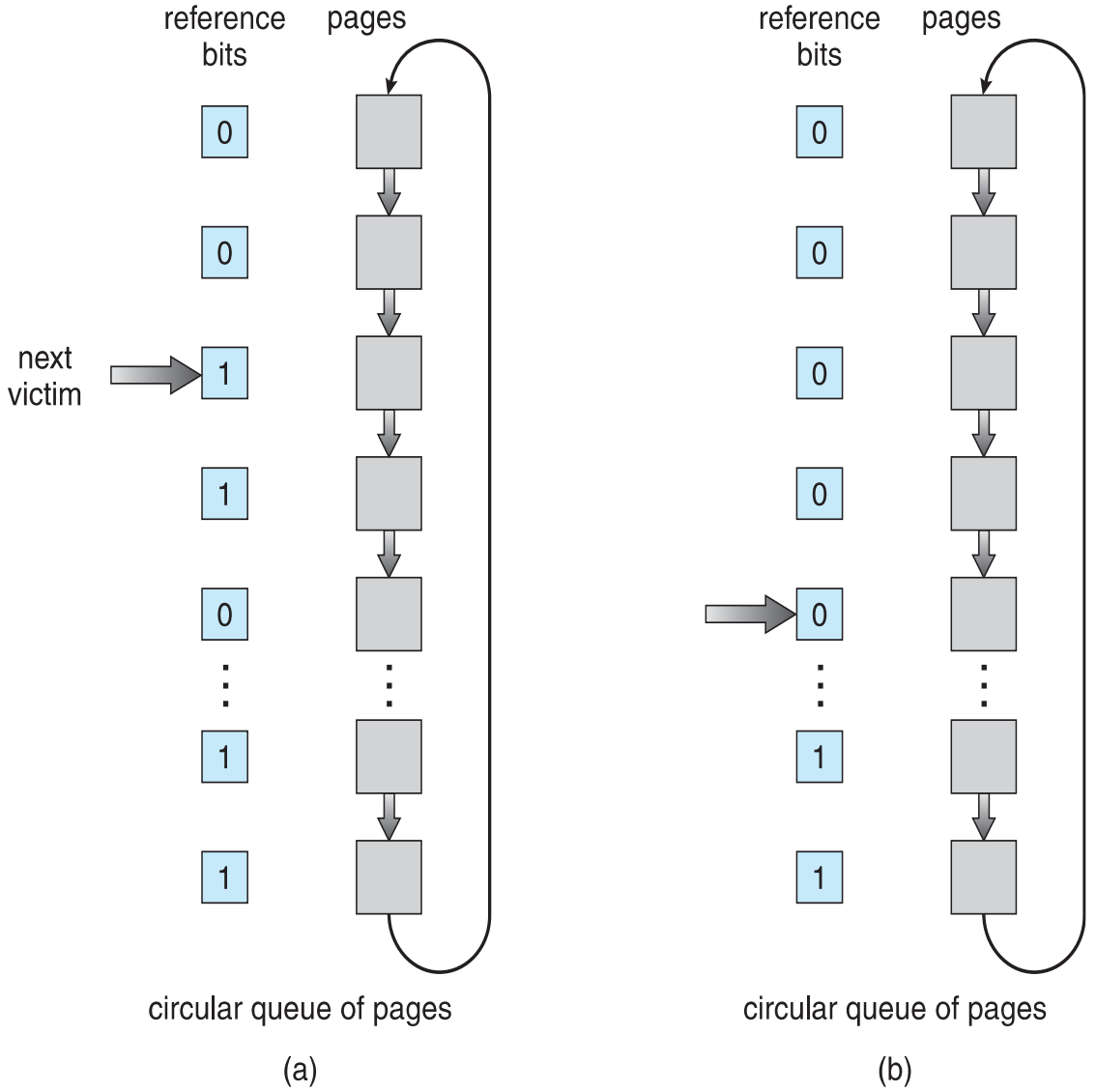
9.4.5 LRU Approximation Algorithms – cont.

Second-chance algorithm

- No shift registers involved, but still requires the hardware reference bit.
- Essentially a FIFO algorithm with modifications – On a page fault:
  - A circular buffer implements the FIFO (looks like a **clock** whose hands point to the next page to examine)
  - On page fault, if currently examined page has:
    - Reference bit = 0 -> replace it.
    - Reference bit = 1, then:
      - set reference bit 0, leave page in memory → give it a ***second chance.***
      - Repeat for next page, subject to same rules



Second-Chance (clock) Page-Replacement Algorithm

LRU Approximation Algorithms – cont.

Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert
- Take ordered pair (reference, modify)

1.(0, 0) neither recently used nor modified – best page to replace

2.(0, 1) not recently used but modified – not quite as good, must swap the page out before replacement

3.(1, 0) recently used but clean – probably will be used again soon

4.(1, 1) recently used and modified – probably will be used again soon and need to write out before replacement

- When page replacement is called for, use the clock scheme but use the four classes to replace page in lowest non-empty class
  - Might need to search circular queue several times

9.4.6 Counting Algorithms

- Keep a counter of the number of references that have been made to each page (since start of the reference string)
  - Since it is required to update the counter in each reference, this is done by the hardware → too complex → Makes this class of algorithms uncommon

- **Least Frequently Used (LFU) Algorithm**: replaces page with smallest count

- **Most Frequently Used (MFU) Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used
  - thus eject page with MFU count assuming it was brought in too long ago and less likely to be needed again.

9.4.7 Page-Buffering Algorithms

- Always **keep a pool of free frames**, thus a frame is readily available when needed instead of being searched for when the page fault takes place:
  - Read page into free frame (swap in)
  - Select victim to evict (using some replacement algorithm)
  - When convenient, evict victim, i.e. **swap out on the background** and add its frame to free pool.
- Possible optimization:
  - keep list of modified pages
  - When backing store otherwise idle, write modified pages there and set to non-dirty

  i.e. try to keep as many pages as possible clean (**clean the pages on the background**)
  → clean pages do not need to be swapped out.
- Another possible optimization:
  - keep free frame content intact and note which pages reside on them (i.e. **don't swap out**)
  - If one of those pages is referenced again before the frame got a new resident, then there is no need to load contents again from disk.
  - Generally useful in reducing penalty if the wrong victim frame was selected.

9.4.8 Applications and Page Replacement

- All previously-discussed algorithms have OS guessing about future page access
  - However, some applications have better knowledge – e.g. databases
  - Additionally, in database systems, the same memory may be buffered twice:
    - OS keeps copy of page in memory as disk I/O buffer
    - Database application keeps page in memory for its own work.
- Instead, the operating system can give direct access to the disk, getting out of the way of the applications
  - **Raw disk** mode allows a database application for example to have full access to a raw disk partition.
  - Bypasses buffering, filename search, locking, etc

9.5 Allocation of Frames

- Each process needs ***minimum*** number of frames
- ***Maximum*** of course is total frames in the system (allocated amongst multiple processes .. according to some criteria)
- Two major allocation schemes
  - fixed allocation
  - Dynamic allocation
- Many variations on these two schemes.

9.5.2 Fixed (static) vs variable (dynamic) Allocations

- **Static** (fixed) allocation - For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Keep some as free frame buffer pool
- **Dynamic**, reacts to changes in the degree of multiprogramming or changes in process sizes.

- **Proportional allocation (vs Equal allocation)** – Allocate according to the size of process
  - s_i = size of process p_i
  - $S = \sum s_i$
  - m = total number of frames
  - a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$m = 62$

$s_1 = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 62 \approx 4$

$a_2 = \frac{127}{137} \times 62 \approx 57$

- **Priority Allocation** - Use a proportional allocation scheme using priorities rather than size

- Alternatively, allocate based on priority + size

9.5.3 Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly (depending on other processes running in the system)
  - Greater throughput so more common

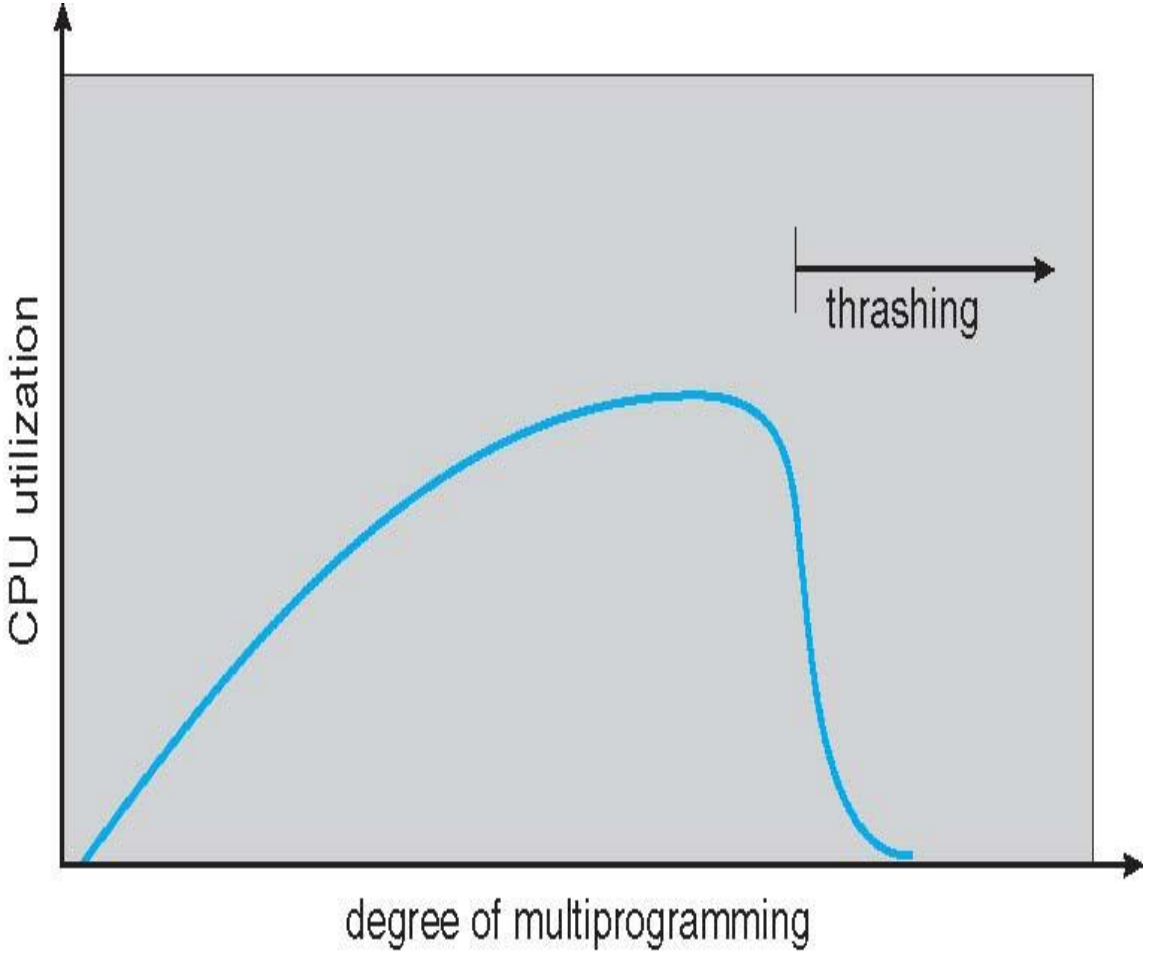
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

9.5.4 Non-Uniform Memory Access

- So far, all memory accessed equally, and we allocated numbers of frames to processes, without specifying their locations.
- Many systems are **NUMA** – speed of access to memory varies
  - Consider system boards containing CPUs and memory, interconnected over a system bus
- Optimal performance comes from allocating memory "close to" the CPU on which the thread is scheduled
  - And modifying the scheduler to schedule all threads of a process on the same system board when possible
  - Solved by Solaris by creating **lgroups**
    - Structure to track CPU / Memory low latency groups
    - Used by scheduler and pager
    - When possible schedule all threads of a process and allocate all memory for that process **within the same lgroup**

9.6 Thrashing

- If a process does not have “enough” frames allocated, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system may think it needs to increase the degree of multiprogramming
    - Another process added to the system

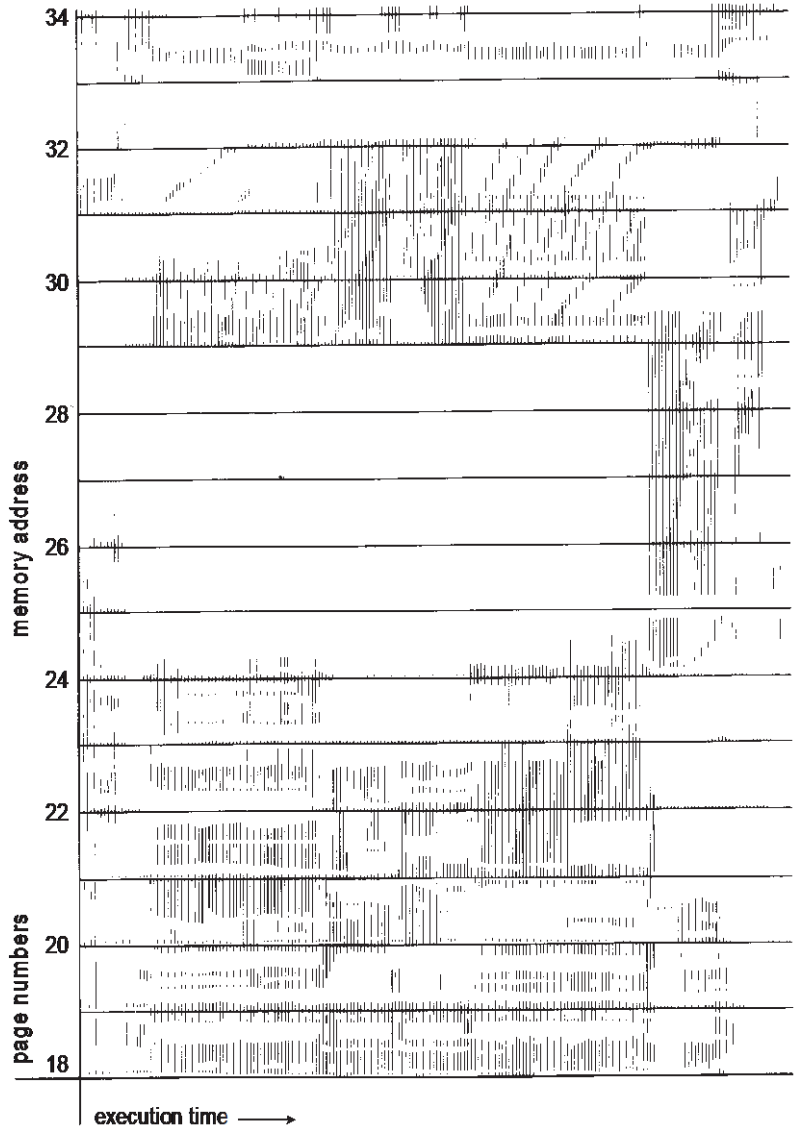


- **Thrashing** ≡ a process is busy swapping pages in and out

Demand Paging and Thrashing

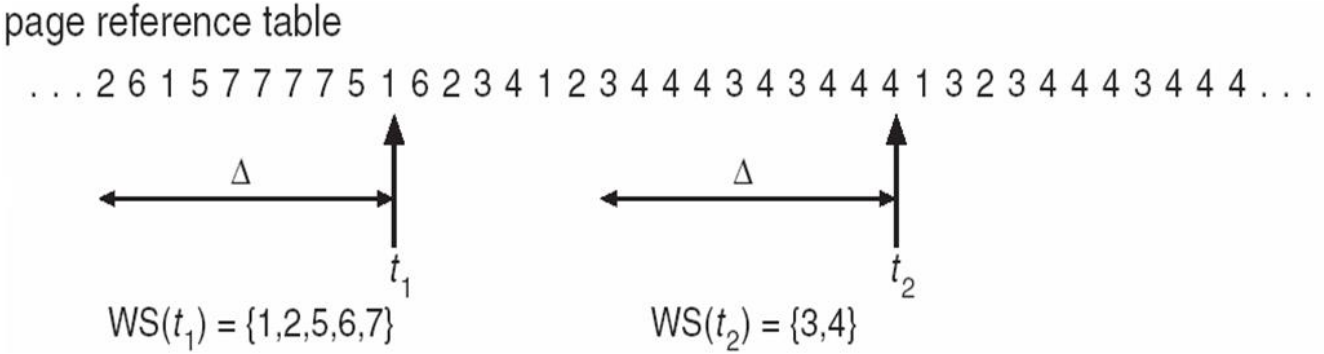
- Why does demand paging work?
  Locality model
  - As a process executes, it moves from locality to another. A locality is a set of pages that are actively used together.
  - Localities may overlap

- Why does thrashing occur?
  Σ size of locality > total memory size allocated
  - Thus allocate the process the frames it needs to avoid thrashing.



Working-Set Model

- **$\Delta \equiv$ working-set window** $\equiv$ a time window specified by a fixed number of page (or memory) references. Note that this **is a sliding window**.
  Example: 10,000 instructions (or 10,000 cycles $\equiv$ 10,000 page references)
- ***WSS_i* (working set size of Process *P_i*)** =
  total number of pages referenced in the most recent Δ (varies over time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program



- $D = \Sigma\ WSS_i \equiv$ total demand frames for all processes
  - Approximation of locality
- if $D > m \Rightarrow$ Thrashing
- Policy if $D >$ m, then suspend or swap out one of the processes
  - **Which process scheduler** does that (short term, medium term or long term scheduler?)

Keeping Track of the Working Set

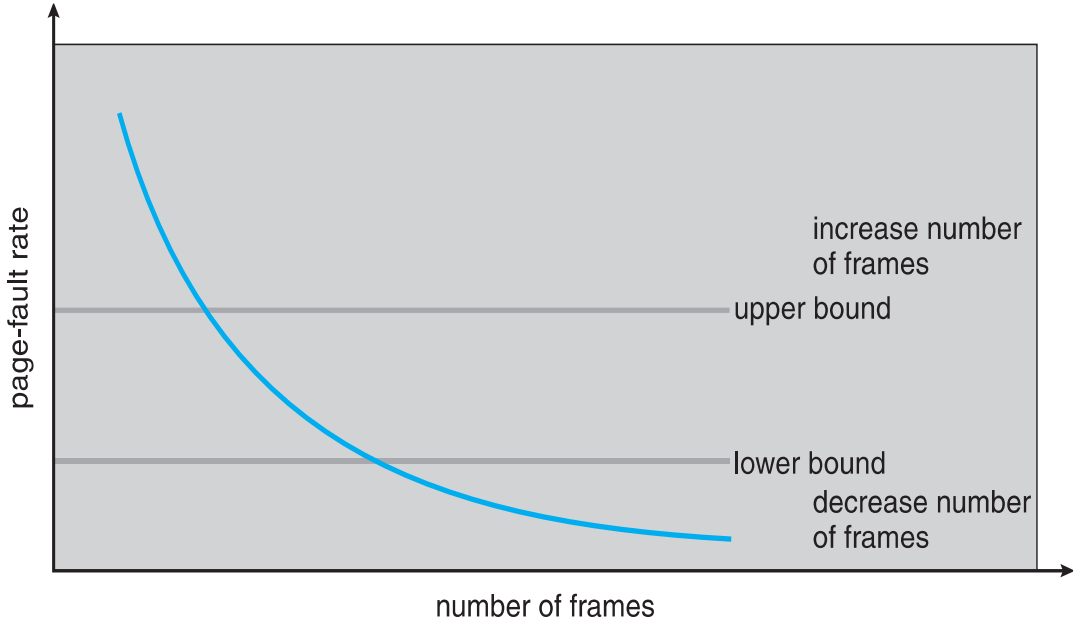
- Approximate with interval timer + a reference bit
- Example: Δ = 10,000
  - Timer interrupts every 5000 time units (hence 2 interrupts)
  - Keep in memory 2 bits for each page (one per interrupt).
  - This is in addition to the reference bit that exists on the page table (one for each page, which is set by the H/W whenever the page is accessed).
  - Whenever a timer interrupt occurs, copy the reference bits on the page table to the corresponding memory-stored reference bits (i.e. one of the 2 bits per page) and set values of all page table reference bits to 0.
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
- Improvement = 8 bits and interrupt every 1250 time units

Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: Δ = 10,000
  - Timer interrupts every 5000 time units (hence 2 interrupts)
  - Keep in memory 2 bits for each page (one per interrupt).
  - This is in addition to the reference bits that exists on the page table (one for each page, which is set by the H/W whenever the page is accessed).
  - Whenever a timer interrupt occurs, copy the reference bits on the page table to the corresponding memory-stored reference bits (i.e. one of the 2 bits per page) and set values of all page table reference bits to 0.
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
  - Working set slides by $\Delta/2$
- Improvement = 8 bits and interrupt every 1250 time units
  - Working set slides by $\Delta/8$

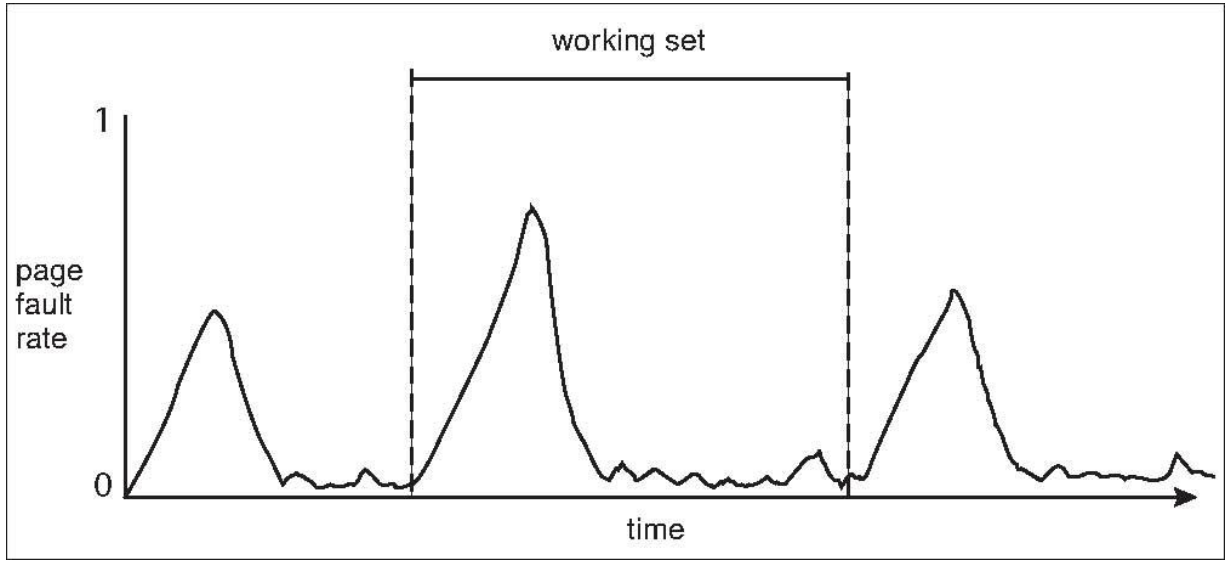
Page-Fault Frequency

- More direct approach than WSS
- Establish “acceptable” **page-fault frequency** (**PFF**) rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
  - Page faults peaks are when a process is changing locality (or working set)
- Peaks and valleys over time

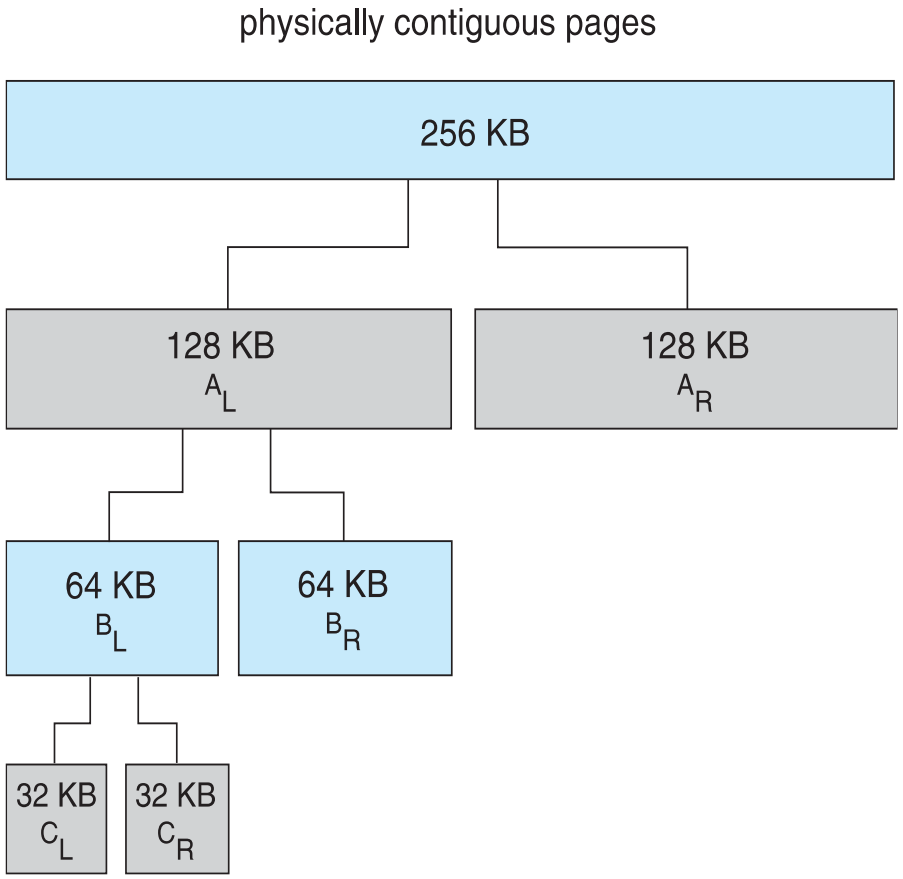


9.8 Allocating Kernel Memory

- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some of the kernel memory allocations needs to be **contiguous**
    - e.g. for device I/O where DMA is used. Some DMA hardware is simple and only operates on contiguous memories (source or destination), whereas other DMA hardware may be complex enough to handle linked lists of memory blocks for I/O transfers.

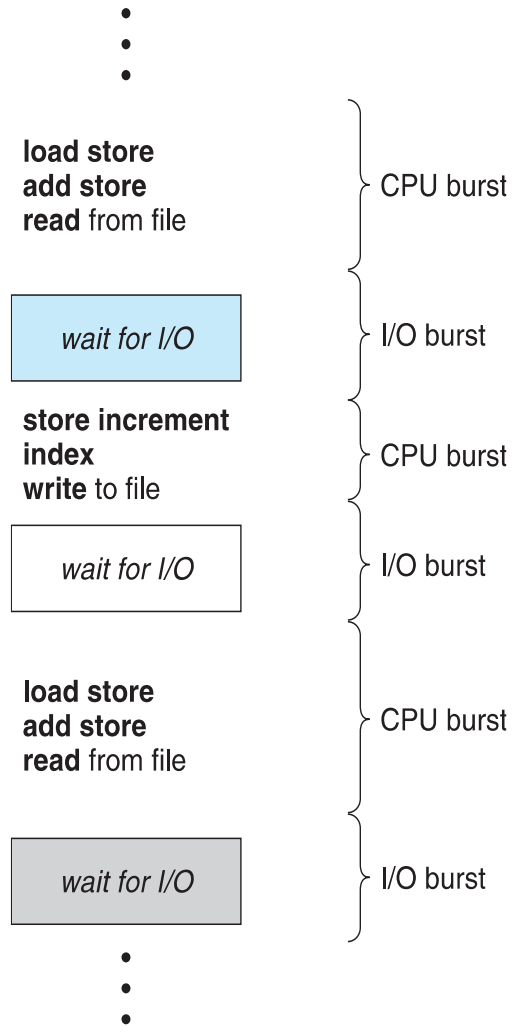
9.8.1 Buddy System

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
    - Continue until appropriate sized chunk available
- For example, assume 256KB chunk available, kernel requests 21KB
  - Split into A_L and A_R of 128KB each
    - One further divided into B_L and B_R of 64KB
      - One further into C_L and C_R of 32KB each – one used to satisfy request
- Advantages:
  - Blocks may be arranged as a tree for quickly finding the requested empty block.
  - Can **quickly coalesce** unused chunks into larger chunk
- Disadvantage - **internal fragmentation**

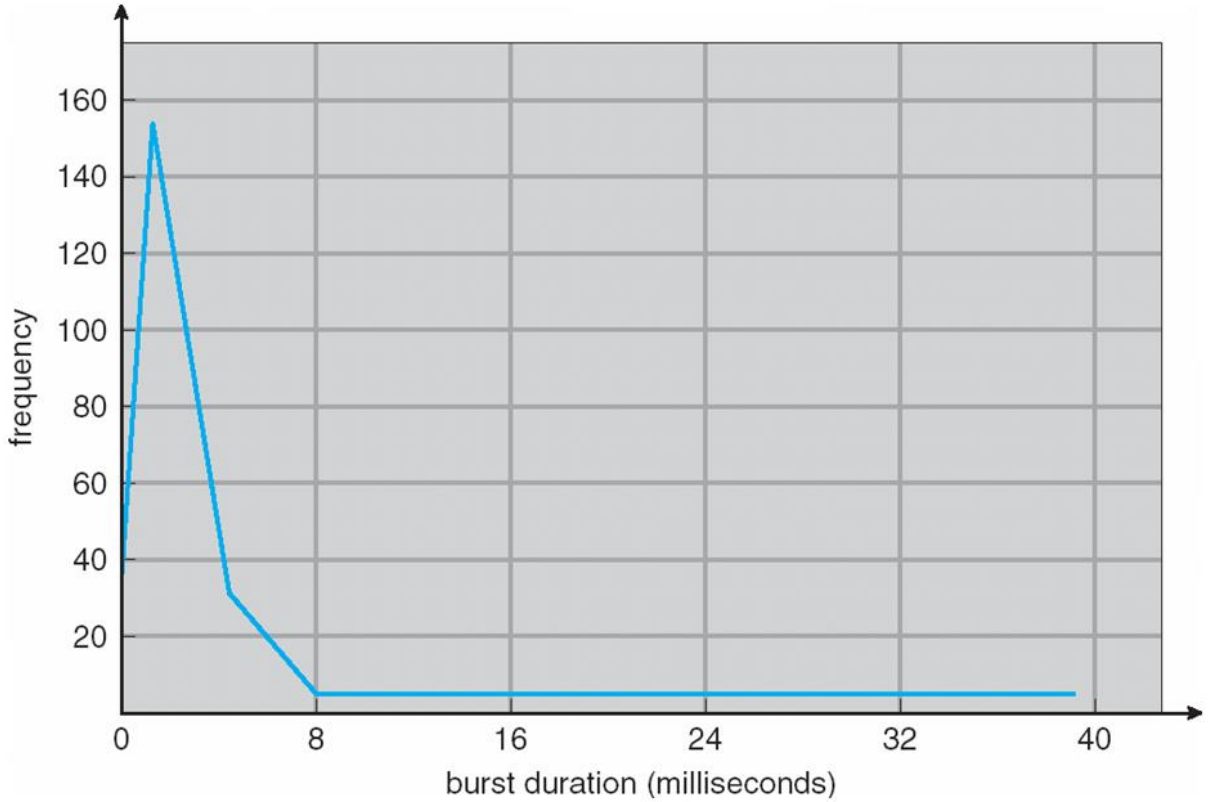


Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern

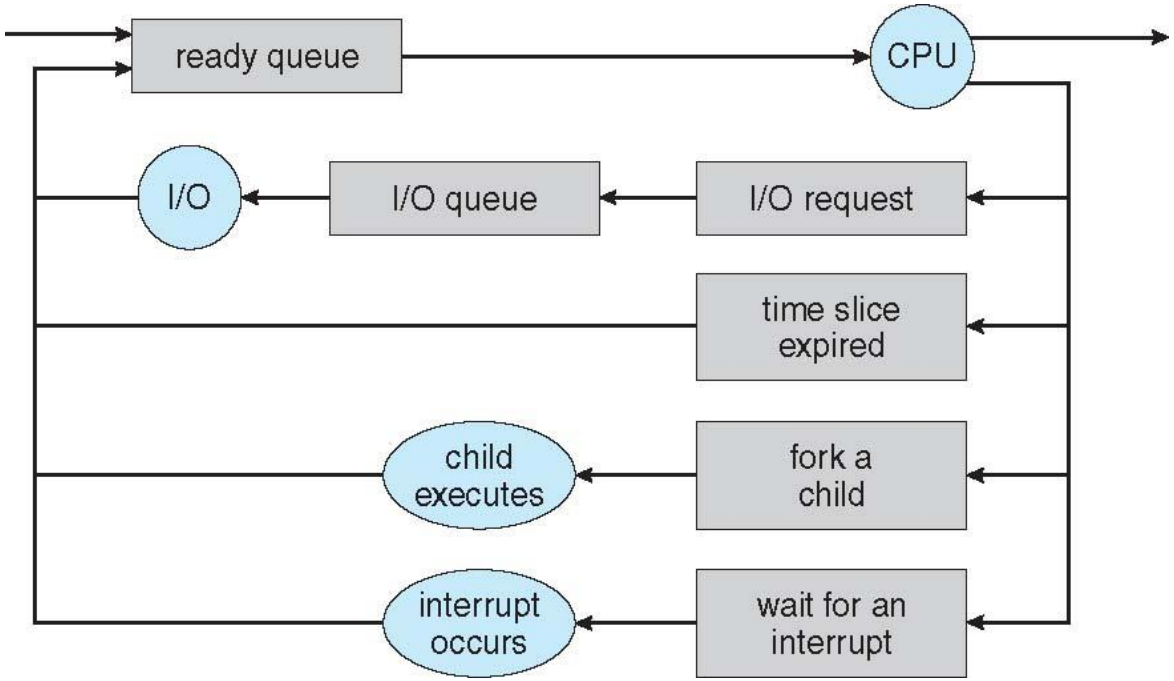


Histogram of CPU-burst Times



CPU Scheduler

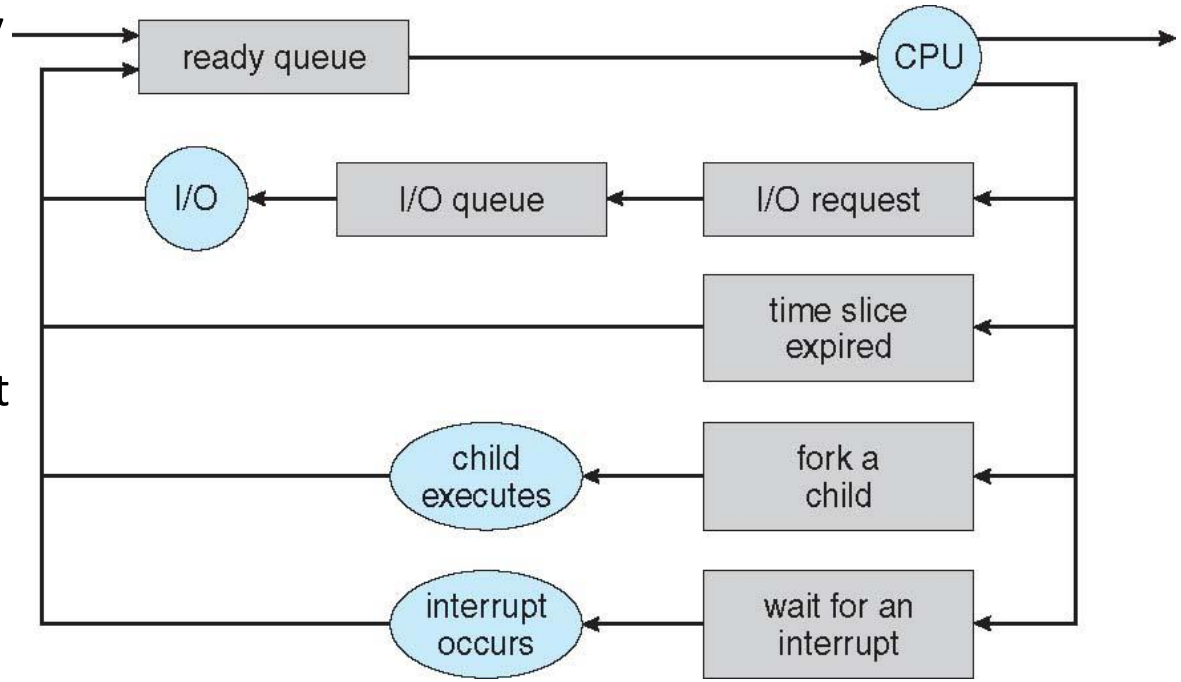
- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- Scheduling may be **non-preemptive** (e.g. FCFS, SJF and priority scheduling). Thus, CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state (i.e. blocked for a resource/device, e.g. a semaphore)
  2. Terminates



CPU Scheduler

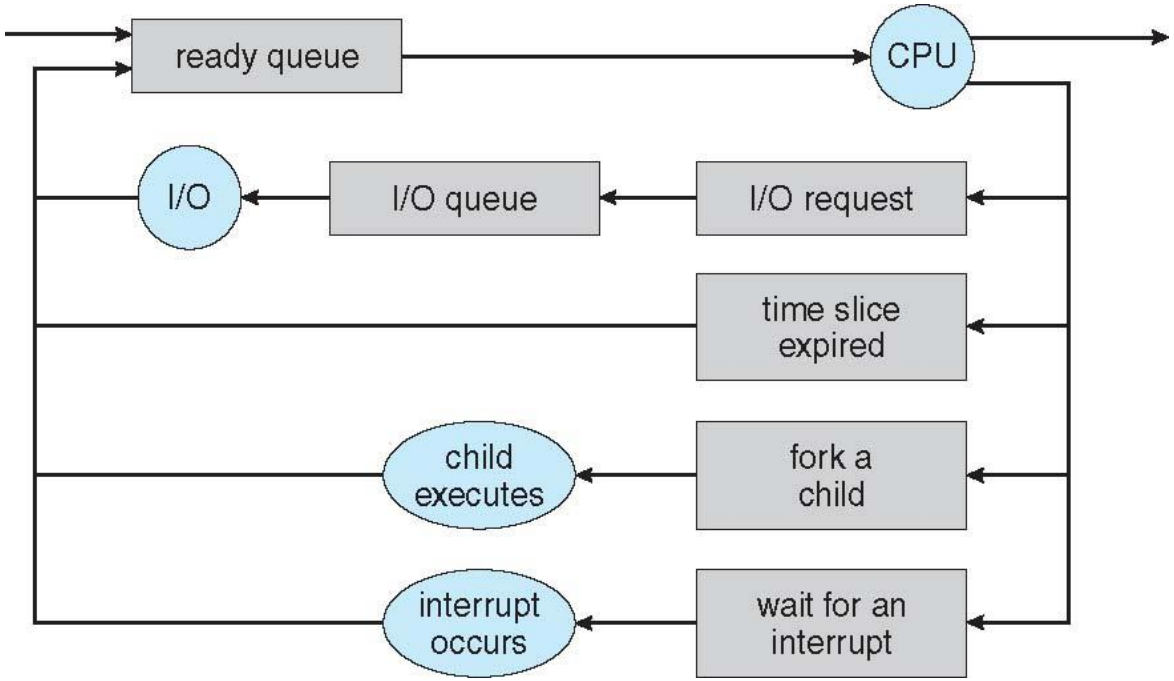
- In **preemptive** scheduling (e.g. round robin), the scheduler may be invoked after an interval time interrupts the CPU or when the process:

1. Switches from running to waiting state (i.e. blocked for a resource/device, e.g. I/O read or a semaphore)
   - Instigated by a system request (software interrupt)
2. Switches from running to ready state
   - Instigated by a timer interrupt (I.e. the timer tick, commonly 10 ms)
3. Switches from waiting to ready state
   - Instigated by a hardware interrupt (e.g. an I/O completion interrupt) OR
   - A system request (e.g. to signal a semaphore)
4. Terminates



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



Scheduling Criteria (metrics targeted for optimization)

- **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
  Completion time – arrival time
- **Waiting time** – amount of time a process has been waiting in the ready queue
  Turnaround time – burst time

- **Response time** – amount of time it takes from when a request was submitted until the first response is produced (not when the response is output)

First- Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:

- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

$$P_2 , P_3 , P_1$$

- The Gantt chart for the schedule is:

- Waiting time for $P_1 = 6; P_2 = 0, P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound (long burst) and many I/O-bound processes (short bursts)

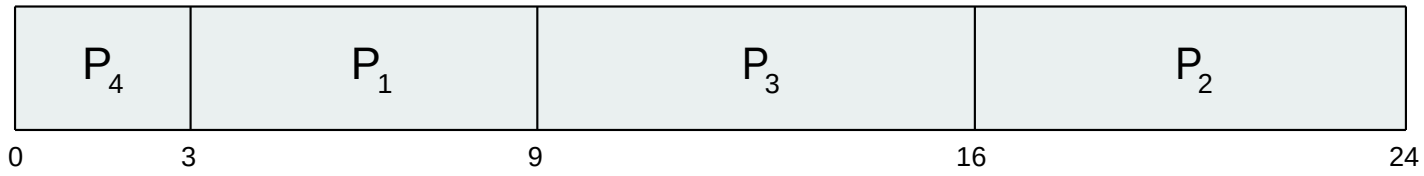
Shortest-Job-First (SJF) Scheduling

- 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 burst length of the next CPU request

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Determining Length of Next CPU Burst

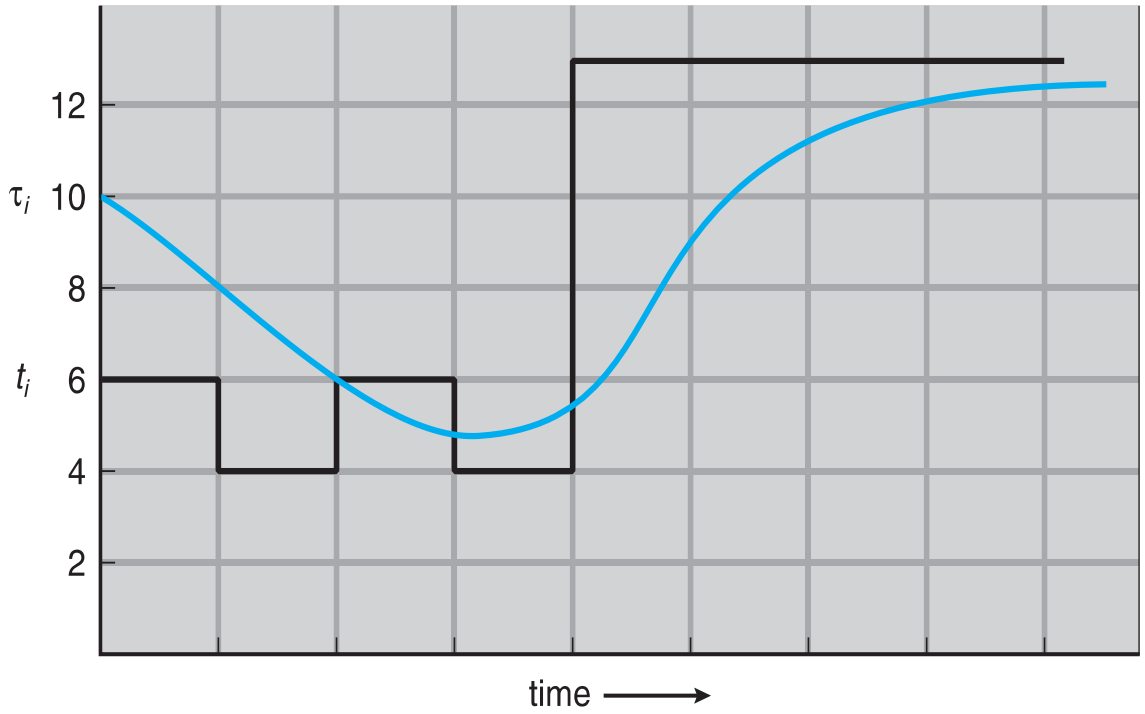
- Can only estimate the length – should be similar to the previous one (of same process of course)
  - Then pick process with shortest predicted next CPU burst

- Can be implemented using the length of previous CPU bursts and exponential averaging

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$ $\tau_{n=1} = \alpha\, t_n + (1-\alpha)\tau_n$.
4. Define :

- Commonly, α (the weight) is set to ½
- Preemptive version called **shortest-remaining-time-first**

Prediction of the Length of the Next CPU Burst



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

Examples of Exponential Averaging

- α =0
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- α =1
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_{n-1} + \ldots$$
$$+ (1 - \alpha)^j \alpha\, t_{n-j} + \ldots$$
$$+ (1 - \alpha)^{n+1} \tau_0$$

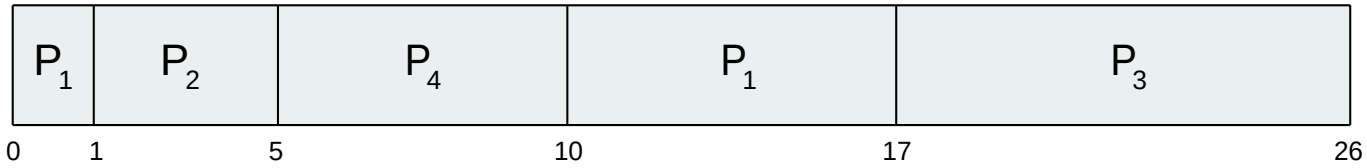
- Since both α and (1 - α) are less than or equal to 1, each successive term has less weight than its predecessor

Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec
- OR (turnaround time – burst time) = [(17-0-8) + (5-1-4) + (26-2-9) + (10-3-5)]/4=6.5

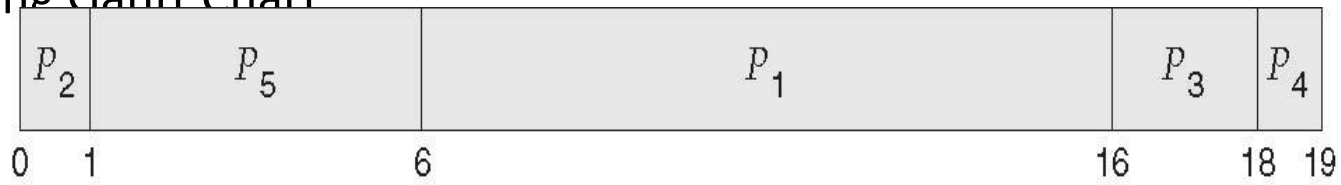
Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - 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

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



- Average waiting time = 8.2 msec

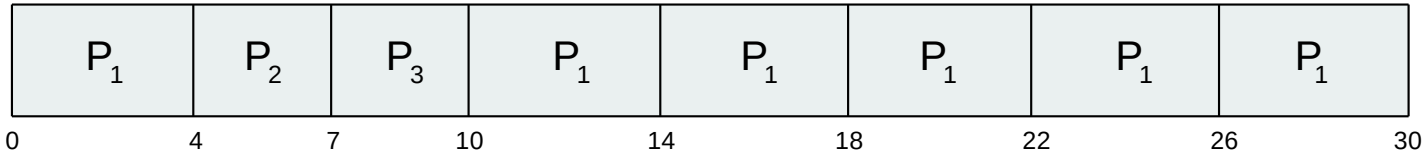
Round Robin scheduling(RR)

- Each process gets a small unit of CPU time (**time quantum** q), usually 1-100 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 must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

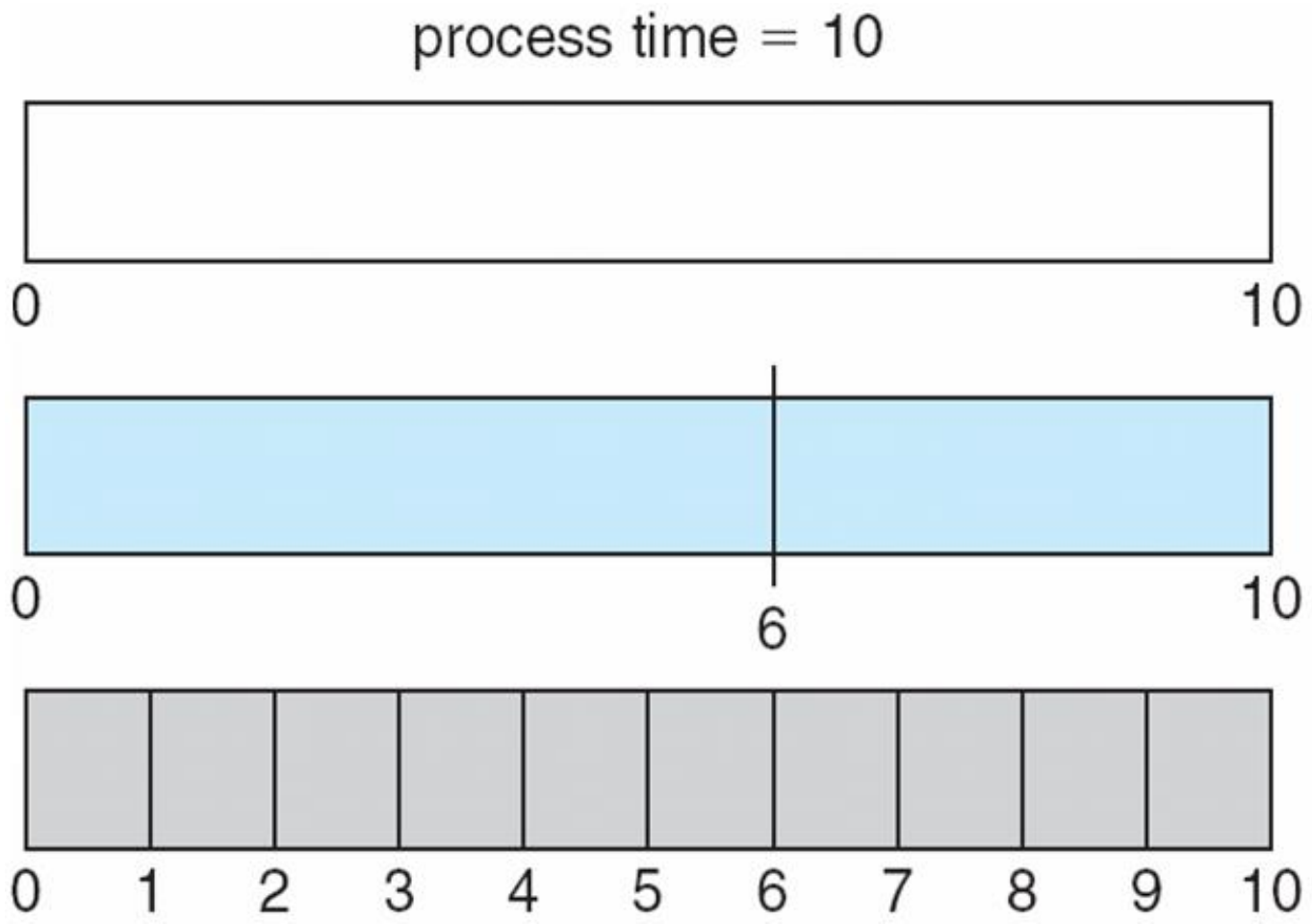
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



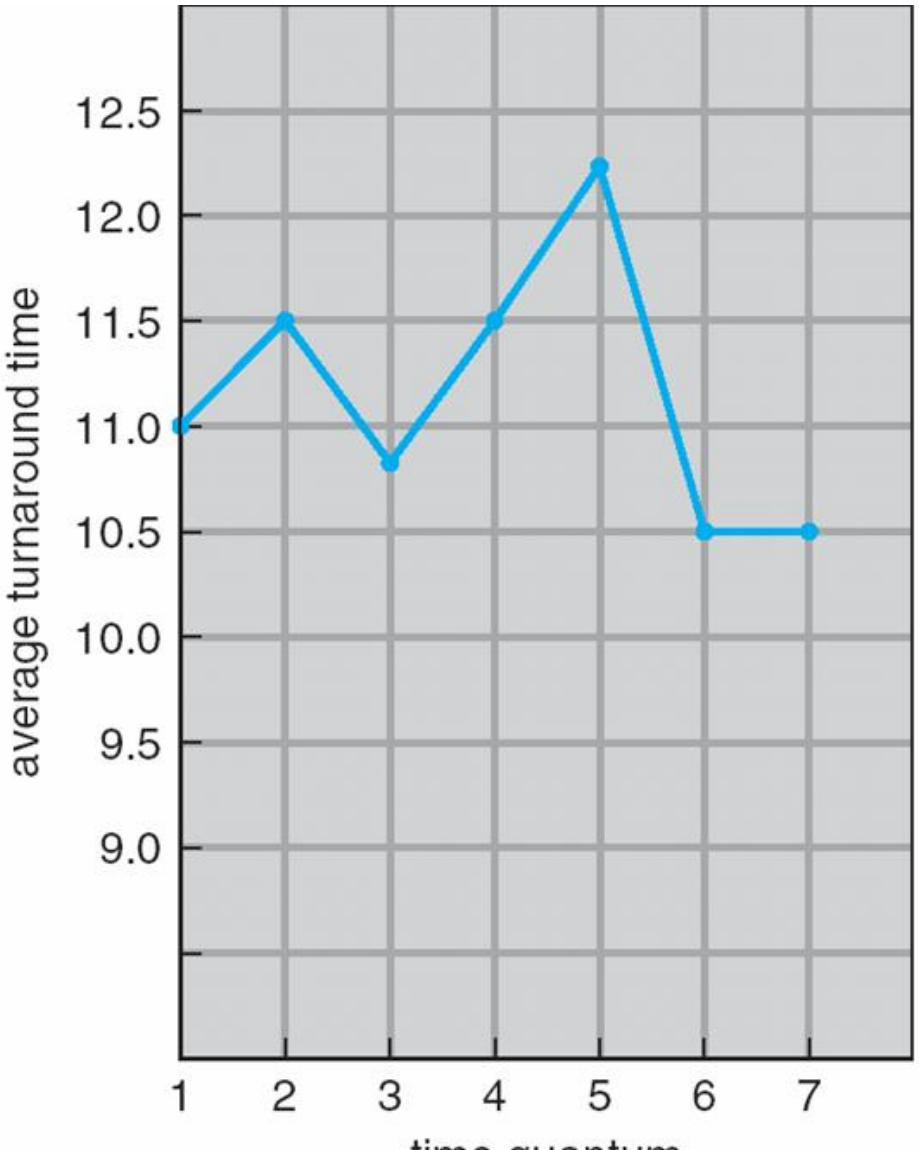
- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
- q usually 1ms to 100ms, context switch < 10 usec

Time Quantum and Context Switch Time



quantum	context switches
12	0
6	1
1	9

Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts
should be shorter than q

Multilevel Queue

- Ready queue is partitioned into separate queues (2 or more), e.g.:
  - **foreground** (interactive)
  - **background** (non-interactive or batch)
- Processes are permanently assigned to a given queue
- Each queue may have its own scheduling algorithm, e.g.:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR, 20% to background in FCFS

Thread Scheduling

- Distinction between user-level and kernel-level threads
- **When threads are supported, threads are scheduled, not processes.**
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on 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 Mac OS X only allow PTHREAD_SCOPE_SYSTEM

Pthread Scheduling API

```
#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("PTHREAD_SCOPE_PROCESS");

      else if (scope == PTHREAD_SCOPE_SYSTEM)

         printf("PTHREAD_SCOPE_SYSTEM");

      else
         fprintf(stderr, "Illegal scope value.\n");

   }
```

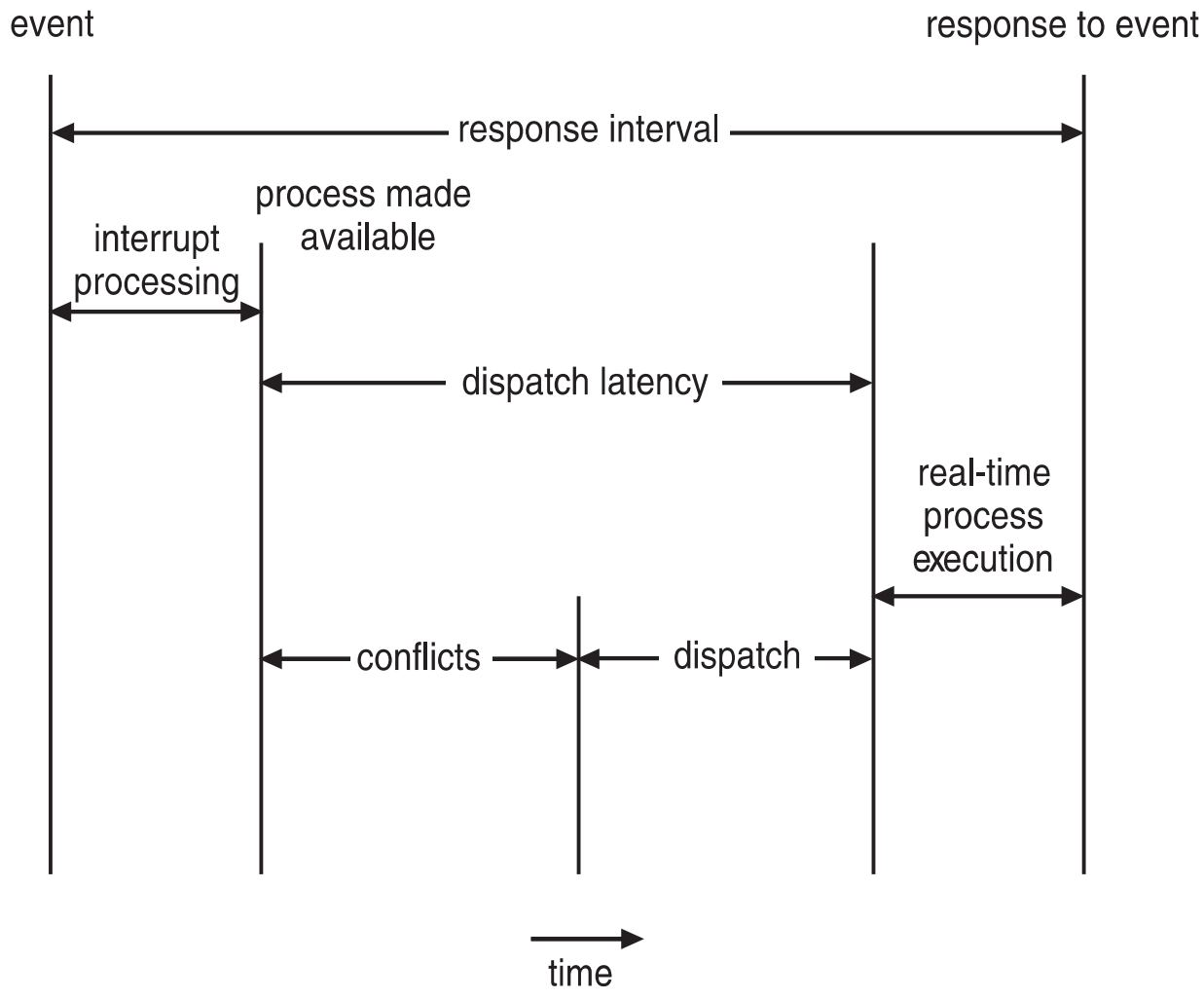
```
   /* set the scheduling algorithm to PCS or
SCS */

   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);

}
```

Real-Time CPU Scheduling

- Can present obvious challenges
- **Soft real-time systems** – guarantees a real-time process will be given preference over other non real-time processes. No guarantee as to when critical real-time process will be scheduled
- **Hard real-time systems** – task must be serviced by its deadline
- Two types of latencies affect performance
  1. **Interrupt latency** – time from arrival of interrupt to start of interrupt service routine (ISR):
     - Hardware **stops current instruction** – may be delayed if interrupts were disabled.
     - Hardware and/or software perform **interrupt context switching** and then start executing the interrupt handler for the particular event or external pin.
  2. **Dispatch latency** – time for scheduler to take current process off CPU and switch to another (**process context switching**)

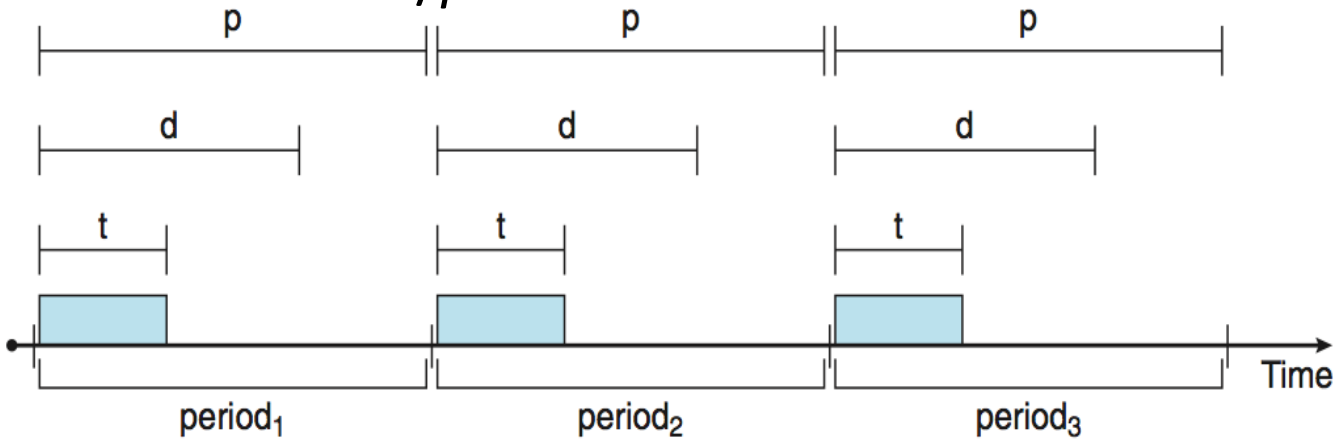
Real-Time CPU Scheduling (Cont.)

- Conflict phase of dispatch latency:
  1. Preemption of any process running in kernel mode
  2. Release by low-priority process of resources needed by high-priority processes



Priority-based Scheduling

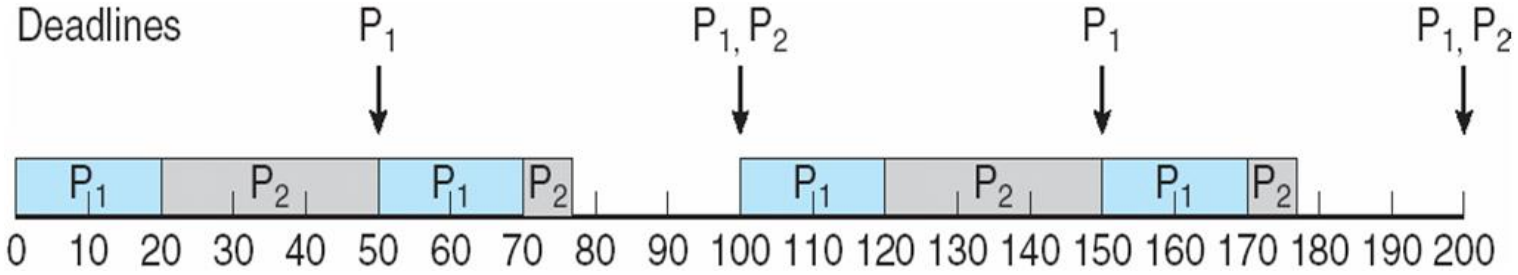
- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But this only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at constant intervals (i.e. they have periodic deadlines)
  - Has processing time *t*, deadline *d,* period *p*
  - $0 \leq t \leq d \leq p$
  - **Rate** of periodic task is $1/p$



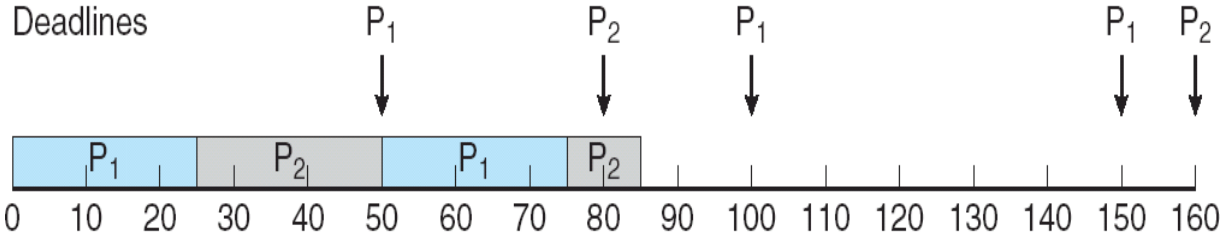
Rate Monotonic Scheduling

- A priority is assigned based on the inverse of its required period (e.g. a task may require processing every 100 time units)
  - Shorter periods = higher priority;
  - Longer periods = lower priority
- Ex: P_1 is assigned a higher priority than P_2.

Rate Monotonic scheduling

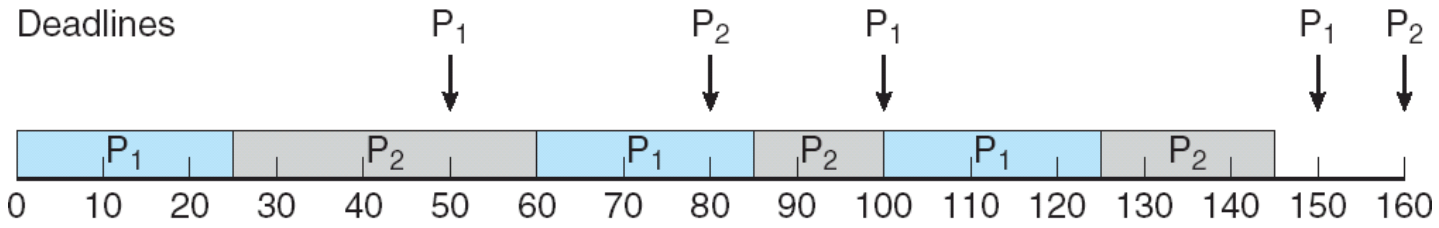


Missed Deadlines with Rate Monotonic Scheduling



Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:
  - The earlier the deadline, the higher the priority;
  - The later the deadline, the lower the priority



Proportional Share Scheduling

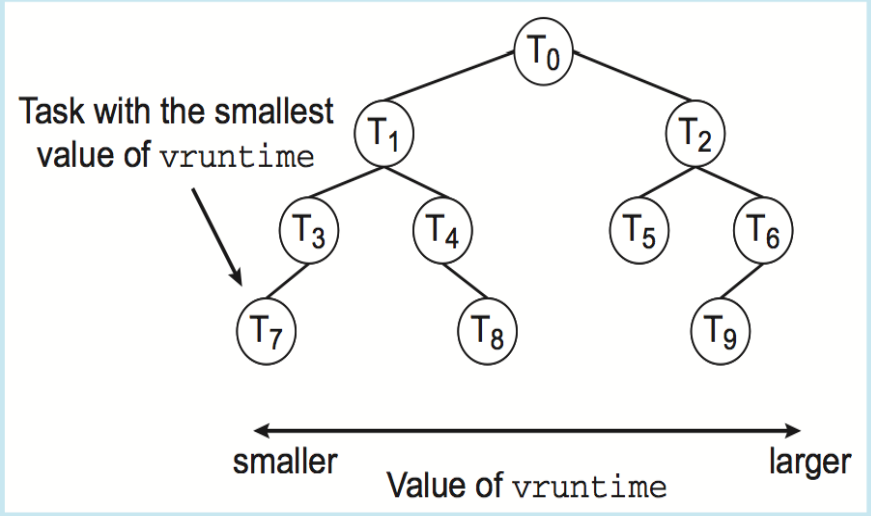
- *T* shares are allocated among all processes in the system
- An application receives *N* shares where $N < T$
- This ensures each application will receive $\boldsymbol{N} / T$ of the total processor time

Linux Scheduling in 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
    1. Default – uses ***Completely Fair Scheduler*** (CFS)
    2. real-time
- Default (non real-time) scheduling:
  - **nice value** from -20 to +19
  - Lower value is higher priority
  - 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 factor
    - Normal default priority yields virtual run time = actual run time
    - To decide next task to run, scheduler picks task with lowest virtual run time.
      - Lower vruntime task preempts a higher vruntime task.

CFS Performance

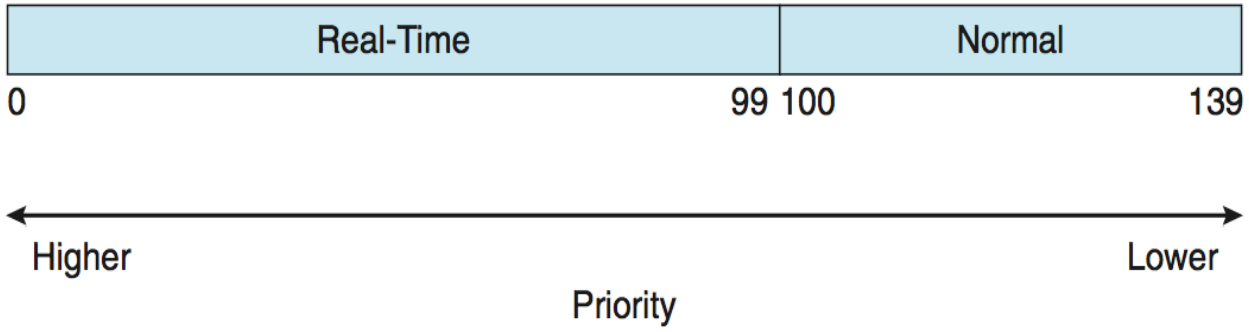
The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(lg\,N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.

Linux Scheduling – POSIX.1b

- 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
- POSIX.1: The nice value is a per-process attribute; that is, the threads in a process should share a nice value.
- Linux: The nice value is a per-thread attribute: different threads in the same process may have different nice values.



Linux/POSIX1.b non real-time classes

- **Non real-time scheduling policies**:
  - **SCHED_OTHER**
    - The **default** scheduling policy for non real-time threads
    - Has static priority of 0.
    - Dynamic priority (nice value) ranges from -20 to 19 (19 has least priority). With each scheduler tick, a thread's dynamic priority decreases → pseudo **round-robin** sharing (this is a logical view, actual implementation uses the CFS/vruntime method)
  - **SCHED_BATCH**
    - Has less static priority than SCHED_OTHER.
    - Used for batch processing, i.e. CPU intensive threads
    - Has a similar dynamic priority scheme as SCHED_OTHER
  - **SCHED_IDLE**
    - For extremely low priority (background) jobs, even lower than SCHED_BATCH with nice values of 19.

Linux/POSIX1.b Real-Time Scheduling

- The POSIX.1b standard - API provides functions for managing real-time threads
- **Real-time scheduling classes**
  - Threads have a priority from 1 to 99
  - Higher priority threads always preempt lower priority threads
  - Always preempts non real-time threads.
  - For real-time threads of equal priority, two policies are offered.

  1. **SCHED_FIFO** – uses a FCFS strategy with a FIFO queue, no time-slicing for threads
  2. **SCHED_RR** - time-slicing occurs for threads of equal priority
- Defines two functions for getting and setting scheduling policy:

```
1.pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)
2.pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)
```

POSIX Real-Time Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
   int i, policy;
   pthread_t tid[NUM_THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* get the current scheduling policy */
   if (pthread_attr_getschedpolicy(&attr,
&policy) != 0)
      fprintf(stderr, "Unable to get
policy.\n");
   else {
      if (policy == SCHED_OTHER)
printf("SCHED_OTHER\n");
      else if (policy == SCHED_RR)
printf("SCHED_RR\n");
      else if (policy == SCHED_FIFO)
printf("SCHED_FIFO\n");
   }
   /* set the scheduling policy */
   if (pthread_attr_setschedpolicy(&attr,
SCHED_FIFO) != 0)
      fprintf(stderr, "Unable to set policy.\n");
   /* 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);
}
```