Lecture 4: Process scheduling and Synchronization

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 – overhead

Scheduling Criteria & Optimization

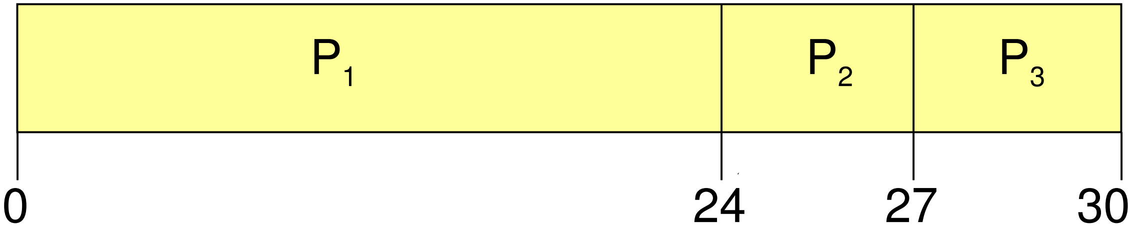
- CPU utilization – keep the CPU as busy as possible
  - Maximize CPU utilization
- Throughput – # of processes that complete their execution per time unit
  - Maximize throughput
- Turnaround time – amount of time to execute a particular process
  - Minimize turnaround time
- Waiting time – amount of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

First-Come, First-Served (FCFS) Scheduling

- Most simple nonpreemptive 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:



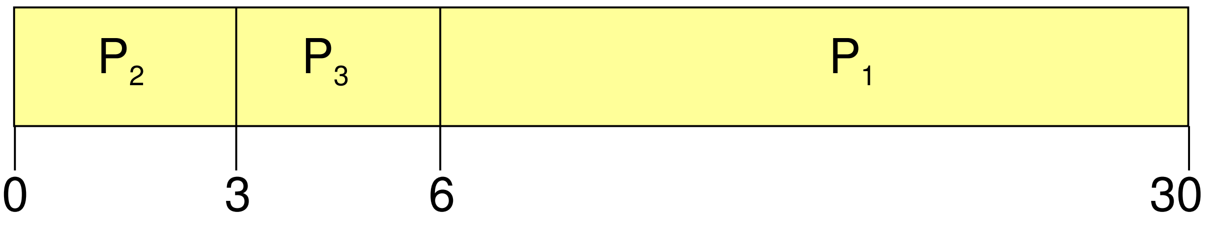
- Waiting time for P_1 = 0; P_2 = 24; P_3= 27
- Average waiting time: (0 + 24 + 27)/3 = 17

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

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
- Two schemes:
  - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
  - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time (SRT)
- SJF is optimal – gives minimum average waiting time for a given set of processes

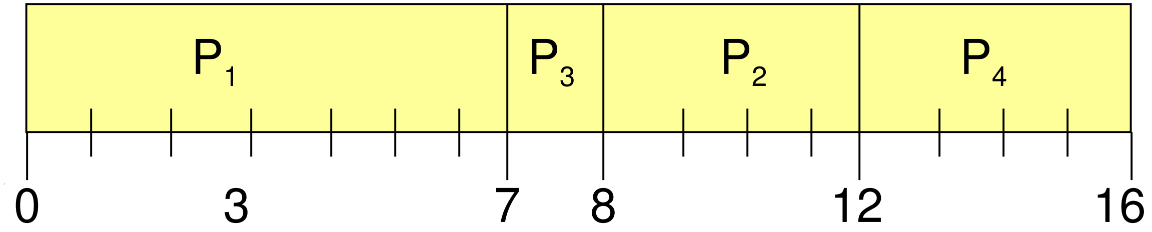
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- Two schemes:
  - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
  - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time (SRT)
- SJF is optimal – gives minimum average waiting time for a given set of processes

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)

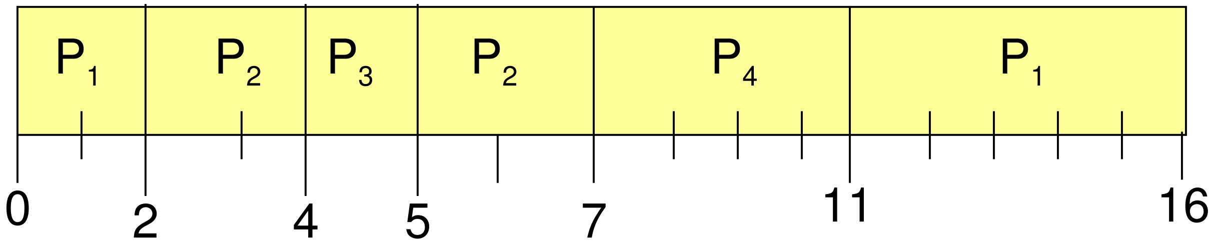


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

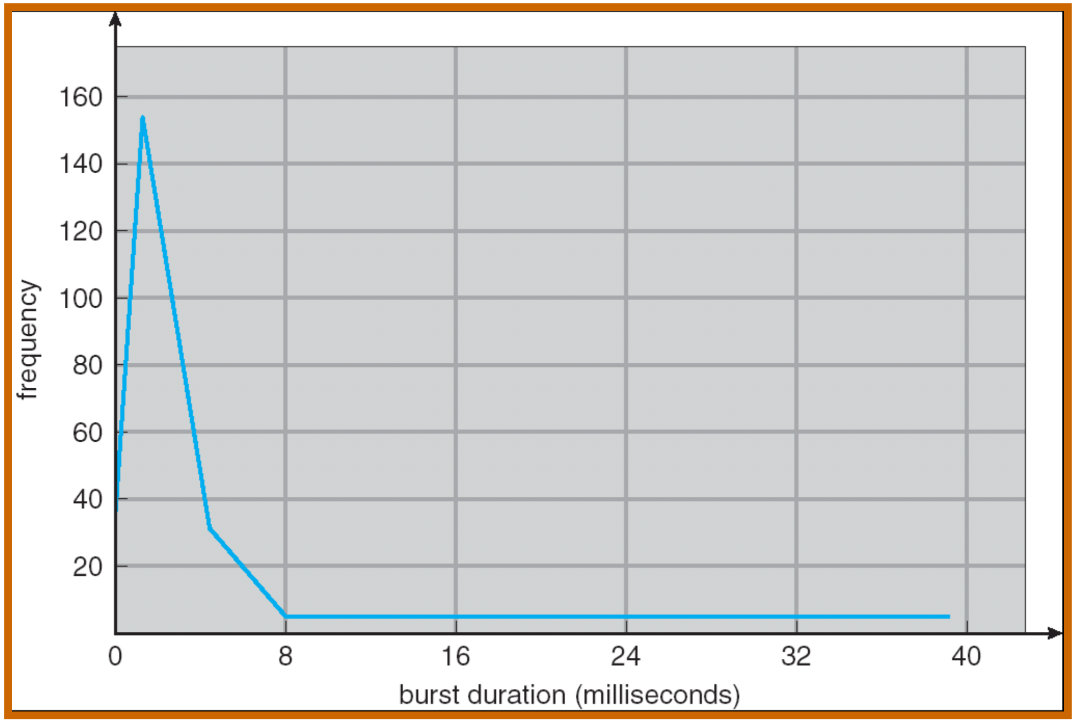
- SJF (preemptive)

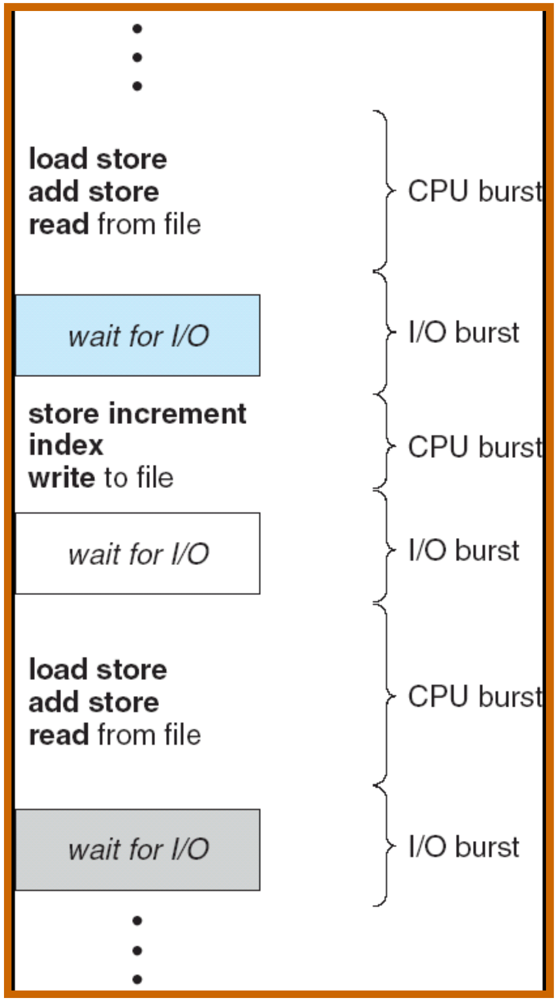


- Average waiting time = (9 + 1 + 0 +2)/4 = 3

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 distribution

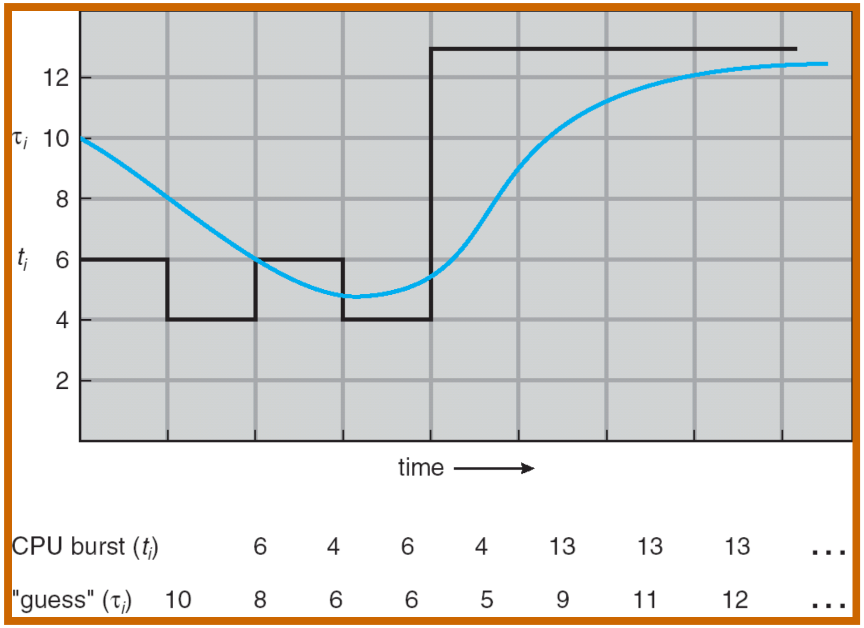




Determining Length of Next CPU Burst

- Can only estimate the length
- Can be done by using the length of previous CPU bursts, using exponential averaging

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



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

Examples of Exponential Averaging

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

$$
\begin{aligned}
\tau_{n+1} = \alpha\, t_n &+ (1 - \alpha)\alpha\, t_{n-1} + \dots \\
&+ (1 - \alpha)^j \alpha\, t_{n-j} + \dots \\
&+ (1 - \alpha)^{n+1} \tau_0
\end{aligned}
$$

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

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 a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ Starvation – low priority processes may never execute (When MIT shut down in 1973 their IBM 7094 - the biggest computer - they found process with low priority waiting from 1967)
- Solution: Aging – as time progresses increase the priority of the process

Round Robin (RR)

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

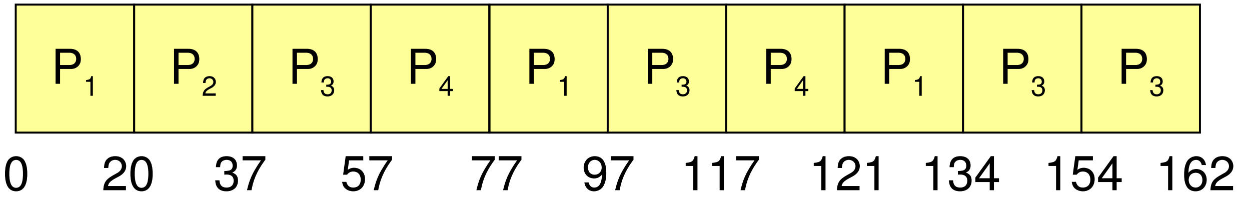
Round Robin (RR)

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

Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:



P_1 P_2 P_3 P_4 P_1 P_3 P_4 P_1 P_3 P_3

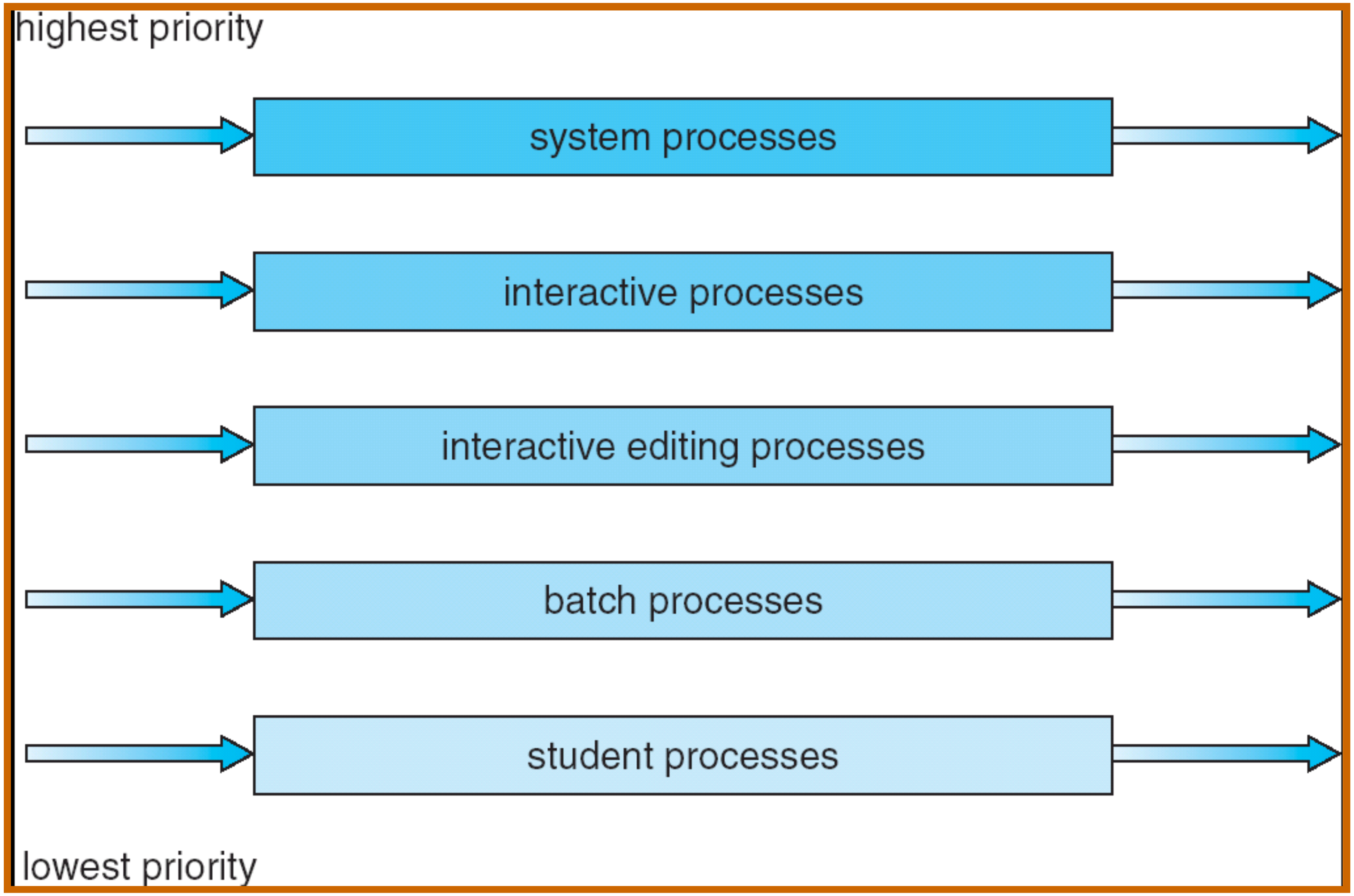
0 20 37 57 77 97 117 121 134 154 162

- Typically, higher average turnaround than SJF, but better *response*

Multilevel Queue

- Ready queue is partitioned into separate queues:
  foreground (interactive)
  background (batch)
- Each queue has its own scheduling algorithm
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Danger 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

Multilevel Queue Scheduling

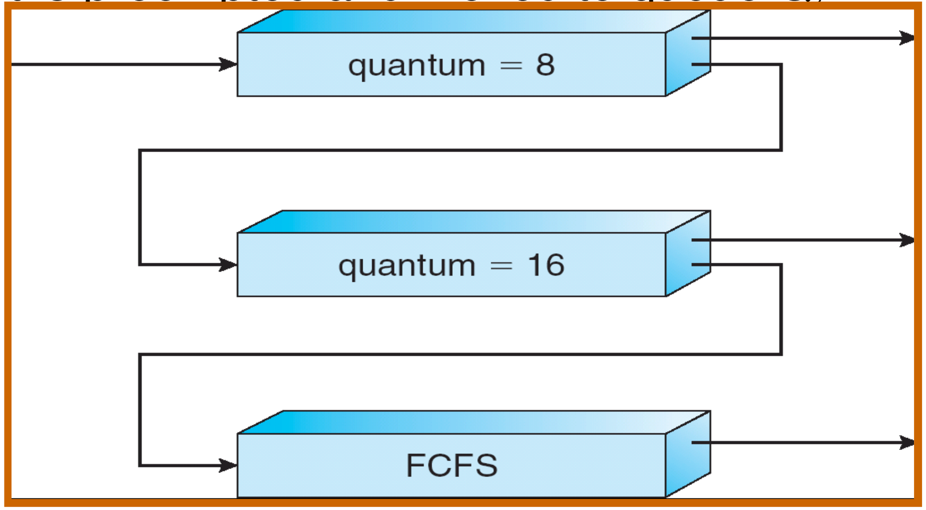


Multilevel Feedback Queue

- A process can move between the various queues; aging can be treated 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:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0. When it gains CPU, job receives 8 milliseconds. If it exhausts 8 milliseconds, job is moved to queue Q_1.
  - At Q_1 the job receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.



Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
  - Multiple-Processor Scheduling has to decide not only which process to execute but also where (i.e. on which CPU) to execute it
- *Homogeneous processors* within a multiprocessor
- *Asymmetric multiprocessing* – only one processor accesses the system data structures, alleviating the need for data sharing
- *Symmetric multiprocessing (SMP)* – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
- *Processor affinity* – process has affinity for the processor on which it has been recently running
  - Reason: Some data might be still in cache
  - *Soft affinity* is usually used – the process can migrate among CPUs

Windows XP Priorities

Priority classes (assigned to each process)

Relative priority within a class

	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

- Relative priority "normal" is a base priority for each class – starting priority of the thread
- When the thread exhausts its quantum, the priority is lowered
- When the thread comes from a wait-state, the priority is increased depending on the reason for waiting
  - A thread released from waiting for keyboard gets more boost than a thread having been waiting for disk I/O

Linux Scheduling

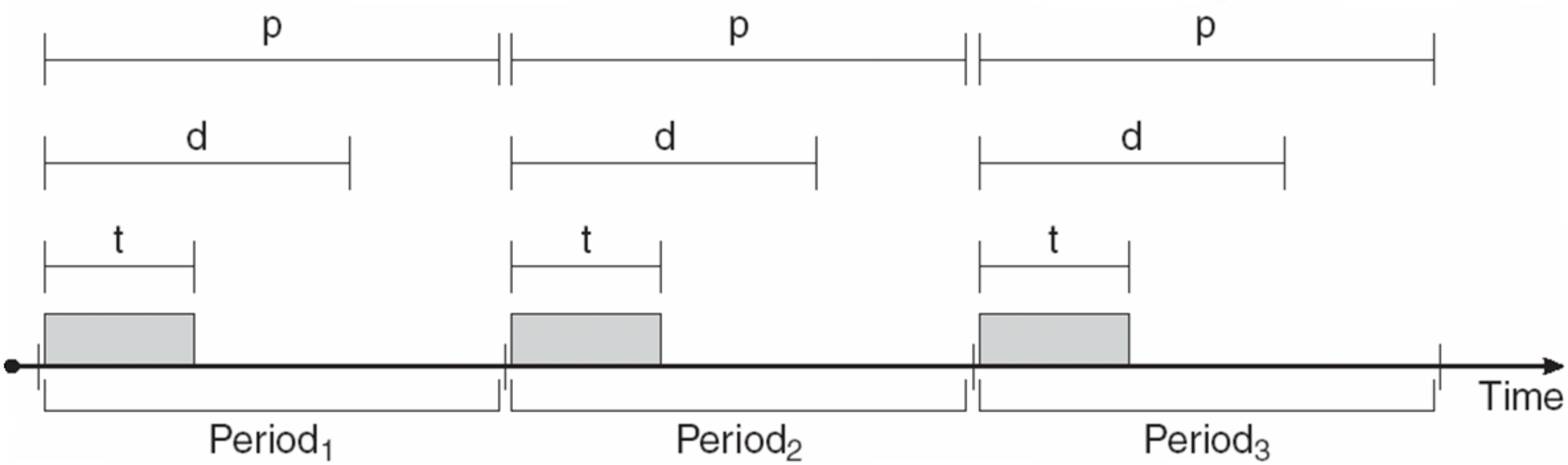
- Two algorithms: time-sharing and real-time
- Time-sharing
  - Prioritized credit-based – process with most credits is scheduled next
  - Credit subtracted when timer interrupt occurs
  - When credit = 0, another process chosen
  - When all processes have credit = 0, recrediting occurs
    - Based on factors including priority and history
- Real-time
  - Soft real-time
  - POSIX.1b compliant – two classes
    - FCFS and RR
    - Highest priority process always runs first

Real-Time Systems

- A real-time system requires that results be not only correct but in time
  - produced within a specified deadline period
- An embedded system is a computing device that is part of a larger system
  - automobile, airliner, dishwasher, ...
- A safety-critical system is a real-time system with catastrophic results in case of failure
  - e.g., airplanes, racket, railway traffic control system
- A hard real-time system guarantees that real-time tasks be completed within their required deadlines
  - mainly single-purpose systems
- A soft real-time system provides priority of real-time tasks over non real-time tasks
  - a “standard” computing system with a real-time part that takes precedence

Real-Time CPU Scheduling

- Periodic processes require the CPU at specified intervals (periods)
- *p* is the duration of the period
- *d* is the deadline by when the process must be serviced (must finish within *d*) – often equal to *p*
- *t* is the processing time



Scheduling of two and more tasks

Can be scheduled if $r = \sum_{i=1}^{N} \frac{t_i}{p_i} \le 1$ (N = number of processes)

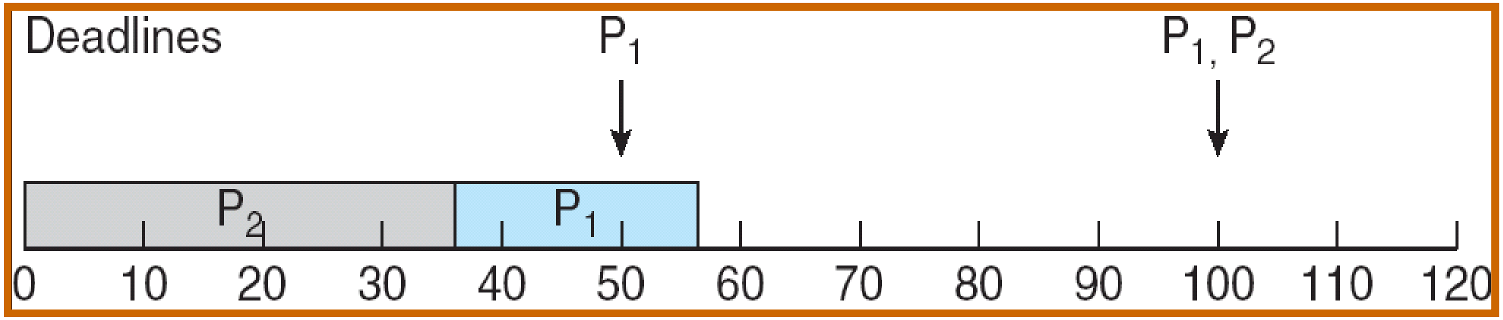
r – CPU utilization

Process P_1: service time = 20, period = 50, deadline = 50

Process P_2: service time = 35, period = 100, deadline = 100

$$r = \frac{20}{50} + \frac{35}{100} = 0.75 < 1 \quad \Rightarrow \quad \text{schedulable}$$

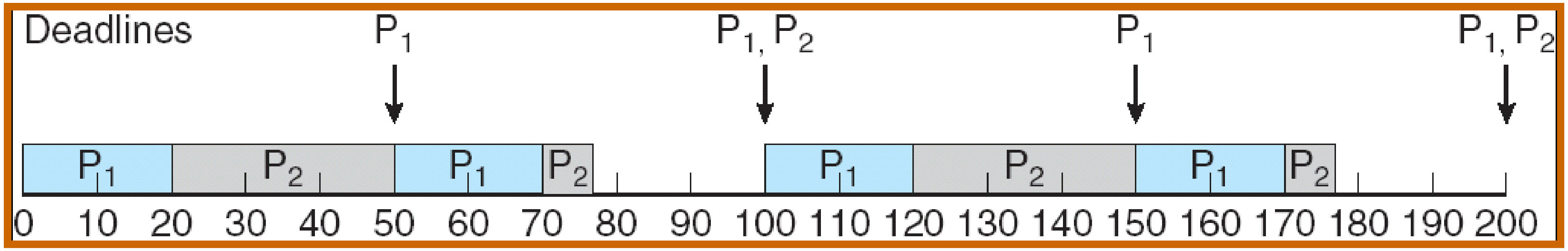
When P_2 has a higher priority than P_1, a failure occurs:



Rate Monotonic Scheduling (RMS)

- A process priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority

- P_1 is assigned a higher priority than P_2.

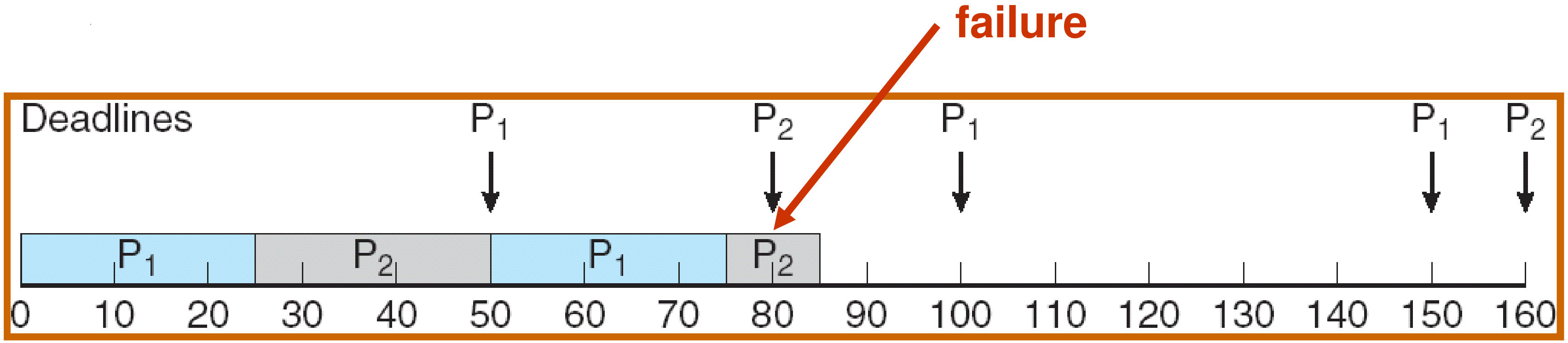


Process P_1: service time = 20, period = 50, deadline = 50

Process P_2: service time = 35, period = 100, deadline = 100

works well

Missed Deadlines with RMS



Process P_1: service time = 25, period = 50, deadline = 50

Process P_2: service time = 35, period = 80, deadline = 80

$$r = \frac{25}{50} + \frac{35}{80} = 0{,}9375 < 1 \Rightarrow \text{schedulable}$$

RMS is guaranteed to work if

$$r = \sum_{i=1}^{N} \frac{t_i}{p_i} \leq N\left(\sqrt[N]{2} - 1\right);$$

N = number of processes

$$\lim_{N \to \infty} N\left(\sqrt[N]{2} - 1\right) = \ln 2 \approx 0.693147$$

sufficient condition

N	$N\left(\sqrt[N]{2}-1\right)$
2	0,828427
3	0,779763
4	0,756828
5	0,743491
10	0,717734
20	0,705298

Analysis of RMS

- Examples

$$3(\sqrt[3]{2}-1)=0{,}7797$$

P	*i*	p_i	t_i	T_i/p_i	$\Sigma(T_i/p_i)$
A	1	7	2	0,286	0,286
B	2	8	3	0,375	0,661
C	3	10	1	0,100	**0,761**

0	1	2	3	4	5	6	7	8	9	10	11
1	1	2	2	2	3		1	1	2	2	

P	*i*	p_i	t_i	T_i/p_i	$\Sigma(T_i/p_i)$
A	1	6	2	0,333	0,333
B	2	8	3	0,375	0,708
C	3	10	1	0,100	**0,808**

0	1	2	3	4	5	6	7	8	9	10	11
1	1	2	2	2	3	1	1	2	2	2	

P	*i*	p_i	t_i	T_i/p_i	$\Sigma(T_i/p_i)$
A	1	4	1	0,250	0,250
B	2	5	1	0,200	0,450
C	3	6	3	0,500	**0,950**

0	1	2	3	4	5	6
1	2	3	3	1	2	Havárie - P_3 nestihnut

P	*i*	p_i	t_i	T_i/p_i	$\Sigma(T_i/p_i)$
A	1	4	1	0,250	0,250
B	2	5	2	0,400	0,650
C	3	20	7	0,350	**1,000**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	2	2	3	1	2	2	3	1	3	2	2	1	3	3	2	1	2		

RMS detailed analysis

- Lehoczky, Sha & Ding [1989]:
  - Sort proceses $\{P_i, i=1\ldots N \mid p_i \le p_{i+1}, i=1\ldots N-1\}$
  - Let's define $W_i(t)$, L_i for i=1...N as

$$W_i(t)=\sum_{j=1}^{i} t_j \lceil t/p_j \rceil, \; L_i(t)=W_i(t)/t,$$

$$L_i=\min_{\{0<t\le t_i\}} L_i(t), \; L=\max_{\{1\le i\le N\}} L_i$$

$W_i(t)$ represents cumulative demands P_1 ... P_i in time $[0, t]$

  - RMS is working correctly if and only if ***L* ≤ 1**.
  - For definition $W_i(t)$ time t is continuos.
  - Lehoczky, Sha and Ding prove, that $W_i(t)$ can be computed only in multiple of periods of all processes
  - Example for $\{p_1 = 4; p_2 = 5; p_3 = 13\}$ it is sufficient to compute $W_i(t)$ and $L_i(t)$ only for

$$t \in \{4,5,8,10,12,13\}$$

Examples RMS

- How to compute $W_i(t)$, L_i, L
  - RMS failed L>1

i	p_i	T_i	T_i/p_i	$\Sigma(T_i/p_i)$
1	4	1	0,250	0,250
2	5	1	0,200	0,450
3	6	3	0,500	**0,950**

$L_i(4)$	$L_i(5)$	$L_i(6)$	L_i	L
0,250	0,400	0,333	0,250	1,167
0,500	0,600	0,667	0,500	
1,250	1,200	1,167	1,167	

  - RMS no failure L<=1

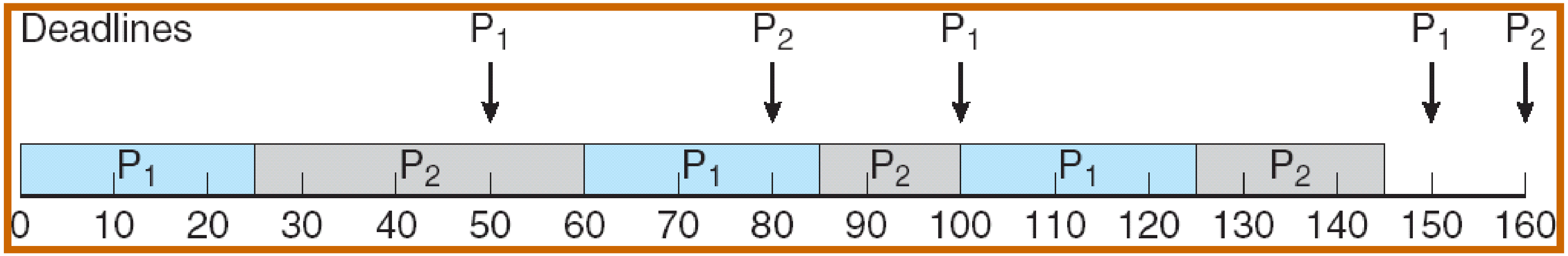
i	p_i	T_i	T_i/p_i	$\Sigma(T_i/p_i)$
1	4	1	0,250	0,250
2	5	2	0,400	0,650
3	20	7	0,350	**1,000**

$L_i(4)$	$L_i(5)$	$L_i(8)$	$L_i(10)$	$L_i(12)$	$L_i(15)$	$L_i(16)$	$L_i(20)$	L_i	L
0,250	0,400	0,250	0,300	0,250	0,267	0,250	0,250	0,250	1,000
0,750	0,800	0,750	0,700	0,750	0,667	0,750	0,650	0,650	
2,500	2,200	1,625	1,400	1,333	1,133	1,188	1,000	1,000	

Earliest Deadline First (EDF) Scheduling

- Priorities are assigned according to deadlines:

  the earlier the deadline, the higher the priority;
  the later the deadline, the lower the priority.



Process P_1: service time = 25, period = 50, deadline = 50

Process P_2: service time = 35, period = 80, deadline = 80

Works well even for the case when RMS failed

PREEMPTION may occur

RMS and EDF Comparison

- **RMS:**
  - Deeply elaborated algorithm
  - Deadline guaranteed if the condition $r \leq N\left(\sqrt[N]{2}-1\right)$ is satisfied (sufficient condition)
  - Used in many RT OS

- **EDF:**
  - Periodic processes deadlines kept even at 100% CPU load
  - Consequences of the overload are unknown and unpredictable
  - When the deadlines and periods are not equal, the behaviour is unknown

Process synchronization

Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience
- Producer-Consumer Problem
  - Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
    - *unbounded-buffer* places no practical limit on the size of the buffer
    - ***bounded-buffer*** assumes that there is a fixed buffer size

Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions
- IPC Implementation
  - Message system – processes communicate with each other without resorting to shared variables
  - Shared memory – not available for distributed systems
- Message system facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., hardware bus, network)
  - logical (e.g., logical properties)

Direct & Indirect Communication

- 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
  - Properties of communication link
    - Links are established automatically
    - A link is associated with exactly one pair of communicating processes
    - Between each pair there exists exactly one link
    - The link may be unidirectional, but is usually bi-directional
- Indirect Communication
  - Messages are directed and received from *mailboxes* (also referred to as *ports*)
    - Each mailbox has a unique *id* and is created by the kernel on request
    - Processes can communicate only if they share a mailbox
  - Properties of communication link
    - Link established only if processes share a common mailbox
    - A link may be associated with many processes
    - Each pair of processes may share several communication links
    - Link may be unidirectional or bi-directional

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send:** the sender blocks until the message is received by the other party
  - **Blocking receive:** the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send:** the sender sends the message and continues executing
  - **Non-blocking receive:** the receiver gets either a valid message or a null message (when nothing has been sent to the receiver)
- Often a combination:
  - Non-blocking send and blocking receive

Producer & Consumer Problem

Message passing:

```
#define BUF_SZ = 20    /* depends on the mailbox size */
typedef struct { … } item_t;
```

Producer:

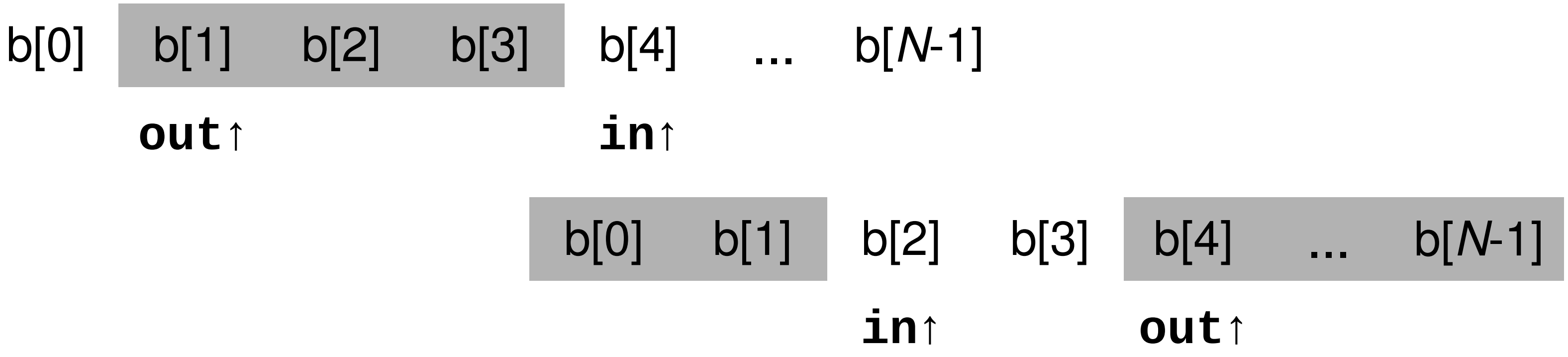
```
void producer() {
    item_t item;
    message m;
    while (1) {
        /* Generate new item */
        receive(consumer, &m);
        /* free slot */
        build_msg(&m, item);
        send(consumer, &m);
    }
}
```

Consumer:

```
void consumer() {
    item_t item;
    message m;
    for (i=0; i<BUF_SZ; i++)
        send(producer, &m);
    while (1) {
        receive(producer, &m)
        item =  extract_item(&m);
        send(producer, &m);
        /* Process nextConsumed */
    }
}
```

Example

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the producer-consumer problem:
  - We have a limited size buffer (*N* items). The *producer* puts data into the buffer and the *consumer* takes data from the buffer
  - We can have an integer count that keeps track of the number of occupied buffer entries. Initially, count is set to 0.
  - It is incremented by the producer after it inserts a new item in the buffer and is decremented by the consumer after it consumes a buffer item



b[0] b[1] b[2] b[3] b[4] ... b[*N*-1]

out↑ **in**↑

b[0] b[1] b[2] b[3] b[4] ... b[*N*-1]

in↑ **out**↑

Producer & Consumer Problem

Shared data:

```
#define BUF_SZ = 20
typedef struct { … } item;
item buffer[BUF_SZ];
int count = 0;
```

Producer:

```
void producer() {
    int in = 0;
    item nextProduced;
    while (1) {
        /* Generate new item */
        while (count == BUF_SZ) ;
            /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUF_SZ;
        count++ ;
    }
}
```

Consumer:

```
void consumer() {
    int out = 0;
    item nextConsumed;
    while (1) {
        while (count == 0) ;
                /* do nothing */
        nextConsumed =  buffer[out];
        out = (out + 1) % BUF_SZ;
        count-- ;
        /* Process nextConsumed */
    }
}
```

- This is a naive solution that does not work

Race Condition

- count++ could be implemented as

```
reg1 = count
reg1 = reg1 + 1
count = reg1
```

- count-- could be implemented as

```
reg2 = count
reg2 = reg2 - 1
count = reg2
```

- Consider this execution interleaving with “count = 5” initially:

S0: producer executes	reg1 = count	{reg1 = 5}
S1: producer executes	reg1 = reg1 + 1	{reg1 = 6}
S2: consumer executes	reg2 = count	{reg2 = 5}
S3: consumer executes	reg2 = reg2 – 1	{reg2 = 4}
S4: consumer executes	count = reg2	{count = 4}
S5: producer executes	count = reg1	{count = 6}

- Variable `count` represents a **shared resource**

Critical-Section Problem

What is a CRITICAL SECTION?

Part of the code when one process tries to access a **particular** resource shared with another process. We speak about a **critical section related to** that **resource**.

1. Mutual Exclusion – If process P_i is executing in its critical section, then no other processes can be executing in their critical sections related to that resource
2. Progress – If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then one of the processes that wants to enter the critical section should be allowed as soon as possible
3. Bounded Waiting – A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the N processes

Critical Section Solution

Critical section has two basic operation: enter_CS and leave_CS

Possible implementation of this operation:

- Only SW at application layer
- Hardware support for operations
- SW solution with supprot of OS

SW solution for 2 processes

- Have a variable *turn* whose value indicates which process may enter the critical section. If *turn* == 0 then P_0 can enter, if *turn* == 1 then P_1 can.

P$_0$

```
while(TRUE) {
        while(turn!=0);   /* wait */
        critical_section();
        turn = 1;
        noncritical_section();
}
```

P$_1$

```
while(TRUE) {
        while(turn!=1);    /* wait */
        critical_section();
        turn = 0;
        noncritical_section();
}
```

- However:
  - Suppose that P_0 finishes its critical section quickly and sets *turn* = 1; both processes are in their non-critical parts. P_0 is quick also in its non-critical part and wants to enter the critical section. As *turn* == 1, it will have to wait even though the critical section is free.
    - The requirement #2 (Progression) is violated
    - Moreover, the behaviour inadmissibly depends on the relative speed of the processes

Peterson’s Solution

- Two processes solution from 1981
- Assume that the LOAD and STORE **instructions are atomic**; that is, cannot be interrupted.
- The two processes share two variables:
  - int turn;
  - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready ($i = 0,1$)

```
j = 1-i;
flag[i] = TRUE;
turn = j;
while ( flag[j] && turn == j);
        // CRITICAL SECTION
flag[i] = FALSE;
```

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Dangerous to disable interrupts at application level
    - Disabling interrupts is usually unavailable in CPU user mode
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this are not broadly scalable
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptible
  - Test memory word and set value
  - Swap contents of two memory words
  - For computers with 2 or more cores – real problem of synchronization
    - Locking bus
    - Cache snooping – synchronization of L1 and L2 caches

TestAndSet Instruction

- Semantics:

```
boolean TestAndSet (boolean *target)
    {
        boolean rv = *target;
        *target = TRUE;
        return rv:
    }
```

- Shared boolean variable lock, initialized to false.
- Solution:

```
    while (TestAndSet (&lock )) ;   // active waiting
         //    critical section
     lock = FALSE;
         //      remainder section
```

Swap Instruction

- Semantics:

```
void Swap (boolean *a, boolean *b)
   {
        boolean temp = *a;
        *a = *b;
        *b = temp:
   }
```

- Shared Boolean variable lock initialized to FALSE; each process has a local Boolean variable key.
- Solution:

```
        key = TRUE;
        while (key == TRUE) {   // waiting
              Swap (&lock, &key );
        }
              //    critical section
         lock = FALSE;
              //    remainder section
```

Synchronization without active waiting

- Active waiting waste CPU
  - Can lead to failure if process with high priority is actively waiting for process with low priority
- Solution: blocking by system functions
  - sleep()   the process is inactive
  - wakeup(process)   wake up process after leaving critical section

```
void producer() {
      while (1) {
            if (count == BUFFER_SIZE)  sleep();                      // if there is no space wait - sleep
            buffer[in] = nextProduced;  in = (in + 1) % BUFFER_SIZE;
            count++ ;
            if  (count == 1) wakeup(consumer);                      // if there is something to consume
      }
}
void consumer() {
      while (1) {
            if (count == 0) sleep();                                  // cannot do anything – wait - sleep
            nextConsumed =  buffer[out];  out = (out + 1) % BUFFER_SIZE;
            count-- ;
            if (count == BUFFER_SIZE-1) wakeup(producer);  // now there is space for new product
      }
}
```

Synchronization without active waiting (2)

- Presented code is not good solution:
  - Critical section for shared variable count and function sleep() is not solved
    - Consumer read count == 0 and then Producer is switch before it call sleep() function
    - Producer insert new product into buffer and try to wake up Consumer because count == 1. But Consumer is not sleeping!
    - Producer is switched to Consumer that continues in program by calling sleep() function
    - When producer fill the buffer it call function sleep() – both processes are sleeping!
- Better solution: Semaphores

Semaphore

- Synchronization tool that does not require busy waiting
  - Busy waiting waists CPU time
- Semaphore *S* – system object
  - With each semaphore there is an associated waiting queue. Each entry in waiting queue has two data items:
    - `value` (of type integer)
    - pointer to next record in the list
  - Two standard operations modify S: wait() and signal()

```
wait(S)   {
            value--;
            if (value < 0) {
              add caller to waiting queue
              block(P);    }
          }
signal(S) {
            value++;
            if (value <= 0) {
              remove caller from the waiting queue
              wakeup(P);    }
            }
```

Semaphore as General Synchronization Tool

- Counting semaphore – the integer value can range over an unrestricted domain
- Binary semaphore – the integer value can be only 0 or 1
  - Also known as mutex lock
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion (mutex)

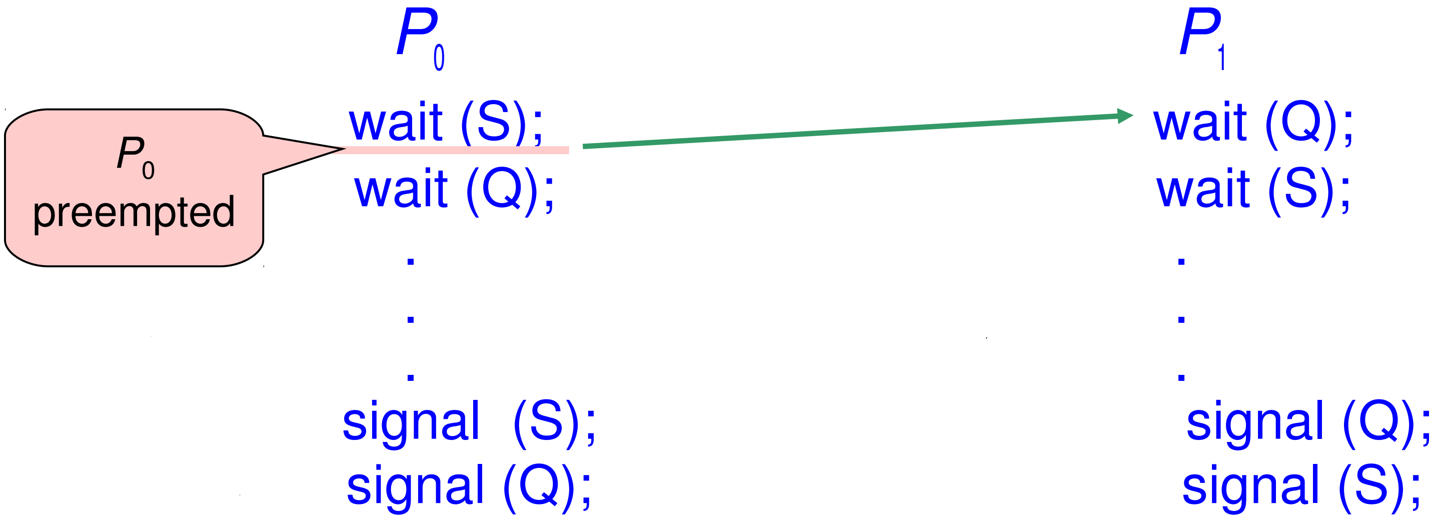
```
Semaphore S;    //  initialized to 1
wait (S);
            Critical Section
signal (S);
```

Spin-lock

- Spin-lock is a general (counting) semaphore using busy waiting instead of blocking
  - Blocking and switching between threads and/or processes may be much more time demanding than the time waste caused by short-time busy waiting
  - One CPU does busy waiting and another CPU executes to clear away the reason for waiting
- Used in multiprocessors to implement short critical sections
  - Typically inside the OS kernel
- Used in many multiprocessor operating systems
  - Windows 2k/XP, Linuxes, ...

Deadlock and Starvation

- **Overlapping critical sections related to different resources**
- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1



```
        P0                         P1
   wait (S);                  wait (Q);
   wait (Q);                  wait (S);
      .                          .
      .                          .
      .                          .
   signal  (S);                signal (Q);
   signal (Q);                signal (S);
```

P_0 preempted

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Classical Problems of Synchronization

- Bounded-Buffer Problem
  - Passing data between 2 processes
- Readers and Writers Problem
  - Concurrent reading and writing data (in databases, ...)
- Dining-Philosophers Problem from 1965
  - An interesting illustrative problem to solve deadlocks
    - Five philosophers sit around a table; they either think or eat
    - They eat slippery spaghetti and each needs two sticks (forks)
    - What happens if all five philosophers
      pick-up their right-hand side stick?
      “They will die of hunger”

Bounded-Buffer Problem using Semaphores

- Three semaphores
  - mutex – for mutually exclusive access to the buffer – initialized to 1
  - used – counting semaphore indicating item count in buffer – initialized to 0
  - free – number of free items – initialized to BUF_SZ

```
void producer() {
   while (1) { /* Generate new item into nextProduced */
               wait(free);
               wait(mutex);
               buffer[in] =  nextProduced; in = (in + 1) % BUF_SZ;
               signal(mutex);
               signal(used);
   }
}

void consumer() {
   while (1) { wait(used);
               wait(mutex);
               nextConsumed = buffer[out];  out = (out + 1) % BUF_SZ;
               signal(mutex);
               signal(free);
               /* Process the item from nextConsumed */
   }
}
```

Readers and Writers

- The task: Several processes access shared data
    - Some processes read the data – readers
    - Other processes need to write (modify) the data – writers
  - Concurrent reads are allowed
    - An arbitrary number of readers can access the data with no limitation
  - Writing must be mutually exclusive to any other action (reading and writing)
    - At a moment, only one writer may access the data
    - Whenever a writer modifies the data, no reader may read it
- Two possible approaches
  - Priority for readers
    - No reader will wait unless the shared data are locked by a writer. In other words: Any reader waits only for leaving the critical section by a writer
    - Consequence: Writers may starve
  - Priority for writers
    - Any ready writer waits for freeing the critical section (by reader of writer). In other words: Any ready writer overtakes all ready readers.
    - Consequence: Readers may starve

Readers and Writers with Readers’ Priority

Shared data

- semaphore wrt, readcountmutex;
- int readcount

Initialization

- wrt = 1; readcountmutex = 1; readcount = 0;

Implementation

Writer:

```
wait(wrt);
....
     writer modifies data
....
signal(wrt);
```

Reader:

```
wait(readcountmutex);
readcount++;
if (readcount==1) wait(wrt);
signal(readcountmutex);

      ... read shared data ...

wait(readcountmutex);
readcount--;
if (readcount==0) signal(wrt);
signal(readcountmutex);
```

Readers and Writers with Writers' Priority

Shared data

- semaphore wrt, rdr, readcountmutex, writecountmutex;
  int readcount, writecount;

Initialization

- wrt = 1; rdr = 1; readcountmutex = 1; writecountmutex = 1;
  readcount = 0; writecount = 0;

Implementation

Reader:

```
wait(rdr);
wait(readcountmutex);
readcount++;
if (readcount == 1) wait(wrt);
signal(readcountmutex);
signal(rdr);

   ... read shared data ...

wait(readcountmutex);
readcount--;
if (readcount == 0) signal(wrt);
signal(readcountmutex);
```

Writer:

```
wait(writecountmutex);
writecount++;
if (writecount==1) wait(rdr);
signal(writecountmutex);
wait(wrt);

           ... modify shared data ...

signal(wrt);
wait(writecountmutex);
writecount--;
if (writecount==0) release(rdr);
signal(writecountmutex);
```

Monitors

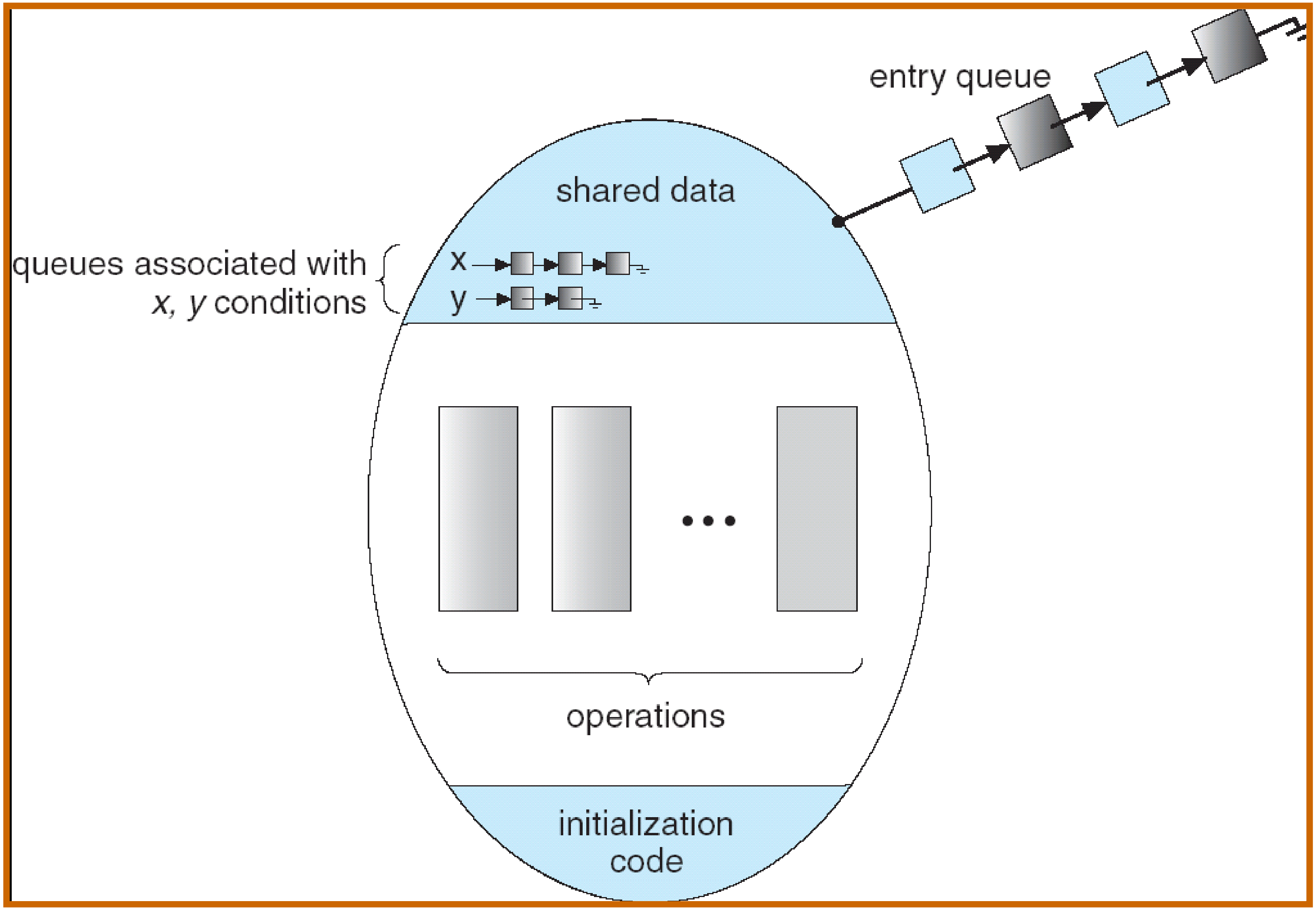
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

```
monitor monitor_name
{
   // shared variable declarations
   condition x, y;  // condition variables declarations
   procedure P1 (…) { …. }
             …
   procedure Pn (…) {……}

   Initialization code ( ….) { … }
             …
   }
}
```

- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended.
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()

Monitor with Condition Variables



Semaphores in Java

- Java is using Monitor for synchronization
- User can define counting semaphore as follows:

```
public class CountingSemaphore {
  private int signals = 1;

  public synchronized void wait() throws InterruptedException{
    while(this.signals == 0) wait();
    this.signals--;
  }

  public synchronized void signal() {
    this.signals++;
    this.notify();
  }
}
```

Synchronization Examples

- Windows XP Synchronization
  - Uses interrupt masks to protect access to global resources on uniprocessor systems
  - Uses spinlocks on multiprocessor systems
  - Also provides dispatcher objects which may act as either mutexes and semaphores
  - Dispatcher objects may also provide events
    - An event acts much like a condition variable
- Linux Synchronization
  - Disables interrupts to implement short critical sections
  - Provides semaphores and spin locks
- Pthreads Synchronization
  - Pthreads API is OS-independent and the detailed implementation depends on the particular OS
  - By POSIX, it provides
    - mutex locks
    - condition variables (monitors)
    - read-write locks (for long critical sections)
    - spin locks

End of Lecture 5

Questions?