Operating Systems CS F372

Lect 30: Threads

BIJU K RAVEENDRAN

Threads

```
#include<pthread.h>
#include<stdio.h>
int value1, value2;
void *runner ( void *param );
void *fact ( void *param );
int main ( int argc, char *argv[ ] )
{           pthread_t tid1,tid2;
            pthread_attr_t attr;
            pthread_attr_init(&attr);
            pthread_create(&tid1, &attr, runner1, argv[1]);
            pthread_create(&tid2, &attr, runner2, argv[2]);
            pthread_join(tid1, NULL);
            pthread_join(tid2, NULL);
            printf( "value1 = %d\t value2 = %d \n", value1,value2);
}
```



```
void *runner ( void *param )
{   int upper = atoi (param);
    int i;  value1=0;
    if ( upper > 0 )
   {
            for ( i=1; i <= upper; i++ )
            { value1 = value1 + i; }
   }
   printf(" New thread runner: value1 = %d, value2 = %d\n",value1, value2);
   pthread_exit(0);
}
```

```
void *fact( void *param )
{   int upper = atoi (param);
    int i;   value2=0;
    if ( upper > =0 )
    {       value2=1;
            for ( i=1; i <= upper; i++ )
            {   value2 = value2 * i; }
    }
 printf(" New thread fact:  value1 = %d, value2 = %d\n",value1, value2);
    pthread_exit(0);
}
```


Thread Related Operations

- Thread creation
- Thread termination
- Thread synchronization
- Thread scheduling
- Thread data management
- Thread / process interaction

Thread Creation

- pthread_create
- extern int pthread_create (pthread_t *tid, __const pthread_attr_t *attr, void *(*__start_routine) (void *), void *arg)
  - Creates a new thread of control that executes concurrently with the calling thread.
  - The new thread applies the function start_routine passing it arg as first argument.
  - The new thread terminates either explicitly, by calling pthread_exit(3), or implicitly, by returning from the start_routine function.
  - The attr argument specifies thread attributes to be applied to the new thread.
  - The attr argument can also be NULL, in which case default attributes are used

Thread Creation

- Return value
  - On success, the identifier of the newly created thread is stored in the location pointed by the thread argument,
  - and a 0 is returned.
  - On error, a non-zero error code is returned.
- Errors
  - EAGAIN not enough system resources to create a process for the new thread.
  - EAGAIN more than PTHREAD_THREADS_MAX threads are already active.

- Setting attributes for threads is achieved by filling a thread attribute object attr of type pthread_attr_t, then passing it as second argument to pthread_create.
- Passing NULL is equivalent to passing a thread attribute object with all attributes set to their default values.
- Thread attribute structure is in /usr/include/bits/pthreadtypes.h

```
#define __SIZEOF_PTHREAD_ATTR_T 56
typedef union
{
 char __size[__SIZEOF_PTHREAD_ATTR_T];
 long int __align;
} pthread_attr_t;
```

Detachstate, Schedpolicy, Sched_param structure, Inheritsched, Scope will be a part of the attribute

Attribute Initialization & Destroy

- extern int pthread_attr_init (pthread_attr_t *attr)
  - Initializes the thread attribute object attr and fills it with default values for the attributes.
  - Attribute objects are consulted only when creating a new thread.
  - The same attribute object can be used for creating several threads. Modifying an attribute object after a call to pthread_create does not change the attributes of the thread previously created.
- extern int pthread_attr_destroy(pthread_attr_t *attr)
  - Destroys a thread attribute object, which must not be reused until it is reinitialized.
  - pthread_attr_destroy does nothing in the LinuxThreads implementation.

Detach State

- Control whether the thread is created in the joinable state (value PTHREAD_CREATE_JOINABLE) or in the detached state (PTHREAD_CREATE_DETACHED).
- Default value: PTHREAD_CREATE_JOINABLE.
- Joinable state
  - Another thread can synchronize on the thread termination and recover its termination code using pthread_join
  - some of the thread resources are kept allocated after the thread terminates, and reclaimed only when another thread performs pthread_join on that thread.
- Detached state
  - The thread resources are immediately freed when it terminates
  - pthread_join cannot be used to synchronize on the thread termination

Detach State

- A thread created in the joinable state can later be put in the detached thread using pthread_detach.
- extern int pthread_attr_setdetachstate (pthread_attr_t *attr, int detachstate)
- extern int pthread_attr_getdetachstate (__const pthread_attr_t *attr, int *detachstate)

Sched Policy

- Select the scheduling policy for the thread: one of SCHED_OTHER (regular, non-realtime scheduling), SCHED_RR (realtime, round-robin) or SCHED_FIFO (realtime, first-in first-out).
- Default value: SCHED_OTHER.
- The real time scheduling policies SCHED_RR and SCHED_FIFO are available only to processes with super user privileges.
- The scheduling policy of a thread can be changed after creation with pthread_setschedparam

Sched Policy

- extern int pthread_attr_setschedpolicy (pthread_attr_t *attr, int policy)
- extern int pthread_attr_getschedpolicy (__const pthread_attr_t *attr, int *policy)

Operating Systems
CS F372

Lect 31: Threads

BIJU K RAVEENDRAN

Attributes

- extern int pthread_attr_init (pthread_attr_t *attr)
- extern int pthread_attr_setdetachstate (pthread_attr_t *attr, int detachstate)
- extern int pthread_attr_getdetachstate (__const pthread_attr_t *attr, int *detachstate)
- extern int pthread_attr_setschedpolicy (pthread_attr_t *attr, int policy)
- extern int pthread_attr_getschedpolicy (__const pthread_attr_t *attr, int *policy)

Sched Param

- Contain the scheduling parameters (essentially, the scheduling priority) for the thread.
- See sched_setparam for more information on scheduling parameters.
- Default value: priority is 0.
- This attribute is not significant if the scheduling policy is SCHED_OTHER; it only matters for the realtime policies SCHED_RR and SCHED_FIFO.
- The scheduling priority of a thread can be changed after creation with pthread_setschedparam

Sched Param

- extern int pthread_attr_setschedparam (pthread_attr_t *attr, __const struct sched_param *param)
- extern int pthread_attr_getschedparam (__const pthread_attr_t *attr, struct sched_param *param)
- See struct sched_param in bits/sched.h

Inheritsched

- Indicate whether the scheduling policy and scheduling parameters for the newly created thread are determined by the values of the schedpolicy and schedparam attributes (value PTHREAD_EXPLICIT_SCHED) or are inherited from the parent thread (value PTHREAD_INHERIT_SCHED).
- Default value: PTHREAD_EXPLICIT_SCHED.
- extern int pthread_attr_setinheritsched (pthread_attr_t *attr, int inherit)
- extern int pthread_attr_getinheritsched (__const pthread_attr_t *attr, int *inherit)

Scope

- Define the scheduling contention scope for the created thread.
- The only value supported in the LinuxThreads implementation is PTHREAD_SCOPE_SYSTEM
  - meaning that the threads contend for CPU time with all processes running on the machine (thread priorities are interpreted relative to the priorities of all other processes on the machine).
- The other value specified by the standard, PTHREAD_SCOPE_PROCESS
  - means that scheduling contention occurs only between the threads of the running process (thread priorities are interpreted relative to the priorities of the other threads of the process, regardless of the priorities of other processes)

Scope

- extern int pthread_attr_setscope (pthread_attr_t *attr, int scope)
- extern int pthread_attr_getscope (__const pthread_attr_t *attr, int *scope)

SetSchedParam

- extern int pthread_setschedparam (pthread_t t_thread, int policy, __const struct sched_param *param)
  - sets the scheduling parameters for the thread t_thread as indicated by policy and param.
  - Policy can be either SCHED_OTHER, SCHED_RR or SCHED_FIFO.
  - param specifies the scheduling priority for the two realtime policies.
- extern int pthread_getschedparam (pthread_t t_thread, int *policy, struct sched_param *param)
  - retrieves the scheduling policy and scheduling parameters for the thread t_thread and store them in the locations pointed to by policy and param, respectively.
- Return value
  - return 0 on success
  - a non-zero error code on error.

Self & equal

- extern pthread_t pthread_self (void)
  - return the thread identifier for the calling thread.
- extern int pthread_equal (pthread_t __thread1, pthread_t __thread2)
  - determines if two thread identifiers refer to the same thread.
  - Returns a non-zero value if thread1 and thread2 refer to the same thread. Otherwise, 0 is returned

Detach

- extern int pthread_detach (pthread_t th)
  - put the thread th in the detached state.
  - applies to threads created in the joinable state, and which needs to be put in the detached state later.
  - After pthread_detach completes, subsequent attempts to perform pthread_join on th will fail.
  - If another thread is already joining the thread th at the time pthread_detach is called, pthread_detach does nothing and leaves th in the joinable state.
- Return value
  - On success, 0 is returned.
  - On error, a non-zero error code is returned.

Exit

- extern void pthread_exit (void *retval)
  - terminates the execution of the calling thread.
  - All cleanup handlers that have been set for the calling thread with pthread_cleanup_push are executed in reverse order.
  - Finalization functions for thread-specific data are then called for all keys that have non- NULL values associated with them in the calling thread (see pthread_key_create).
  - Finally, execution of the calling thread is stopped.
  - The retval argument is the return value of the thread. It can be consulted from another thread using pthread_join.
- Return value
  - The pthread_exit function never returns.

Join

- extern int pthread_join (pthread_t th, void **__thread_return)
  - suspends the execution of the calling thread until the thread identified by th terminates, either by calling pthread_exit or by being cancelled.
  - If thread_return is not NULL, the return value of th is stored in the location pointed to by thread_return.
  - The return value of th is either the argument it gave to pthread_exit, or PTHREAD_CANCELED if th was cancelled.
  - The joined thread th must be in the joinable state
  - When a joinable thread terminates, its memory resources (thread descriptor and stack) are not deallocated until another thread performs pthread_join on it.
  - It is must to call pthread_join once for each joinable thread created to avoid memory leaks.

Join

- At most one thread can wait for the termination of a given thread.
- Calling pthread_join on a thread th on which another thread is already waiting for termination returns an error.

- Cancellation
  - pthread_join is a cancellation point.
  - If a thread is canceled while suspended in pthread_join, the thread execution resumes immediately and the cancellation is executed without waiting for the th thread to terminate.
  - If cancellation occurs during pthread_join, the th thread remains not joined.
- Return value
  - On success, the return value of th is stored in the location pointed to by thread_return, and 0 is returned.
  - On error, a non-zero error code is returned.

Operating Systems CS F372

Lect 32: Threads

BIJU K RAVEENDRAN

Threading Issues

- The fork and exec system calls
  - If one thread in a system calls fork()
    - The new process duplicates all threads
    - The new process duplicates only the calling thread.

Linux Threads

- Linux refers to them as *tasks* rather than *threads*
- Thread creation is done through **clone()** system call
- **clone()** allows a child task to share the address space of the parent task (process)
- Flags control behavior

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Thread Pools

- Create a number of threads in a pool where they await work
- Unlimited threads could exhaust system resources and one solution for this is pooling.
  - Create a number of threads at process startup and place them into a pool. When the server require the thread it is allotted and after completion of task it will be back on pool.
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
  - Separating task to be performed from mechanics of creating task allows different strategies for running task

Thread Cancellation

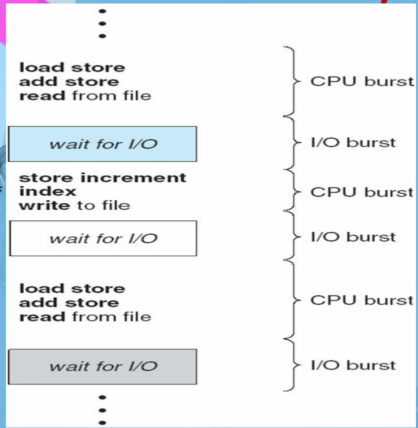
- Terminating a thread before it has finished
- In Linux, thread cancellation is handled through signals
- Two general approaches:
  - **Asynchronous cancellation**
    - Terminates the target thread immediately
  - **Deferred cancellation** [default type]
    - Allows the target thread to periodically check if it should be cancelled
    - Cancellation only occurs when thread reaches cancellation point, then cleanup is invoked

CPU Scheduling

Schedulers

- **Long-term scheduler** (or job scheduler)
  - New to Ready
  - Controls the degree of multiprogramming
- **Short-term scheduler** (or CPU scheduler or Dispatcher)
  - Ready to Running
  - Most frequently executed scheduler
- **Mid-term scheduler**
  - To Ready suspend / Blocked Suspend
  - Part of swapping function
  - Manages the degree of multiprogramming

CPU and I/O Bursts



Processes can be described as either:

I/O-bound process – spends more time doing I/O than computations, many short CPU bursts

CPU-bound process – spends more time doing computations; few very long CPU bursts

CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- 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**
- All other scheduling is **preemptive**

Operating Systems
CS F372

Lect 33:
CPU Scheduling

BIJU K RAVEENDRAN

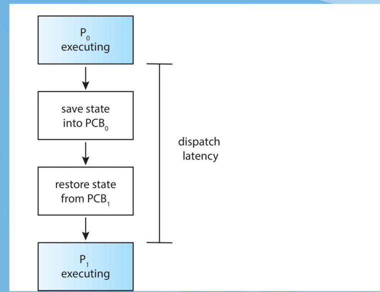
CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- 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**
- All other scheduling is **preemptive**

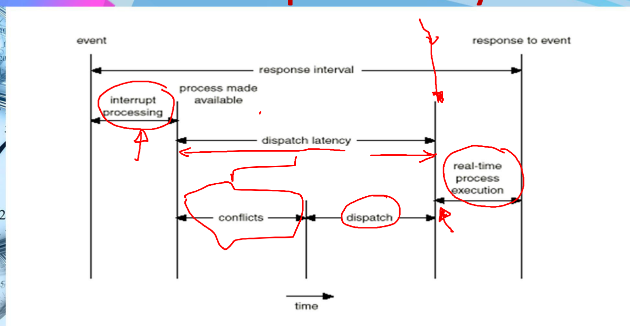
Dispatcher

- 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

Dispatch Latency



Dispatch Latency



CPU Scheduling Criteria

- **CPU utilization**
  - Percentage of time that a processor is busy
- **Throughput**
  - Number of processes that complete their execution per time unit
- **Turnaround time**
  - Interval of time between the submission of a process and its completion.
  - Actual execution time + time spent waiting for resources including the processor

CPU Scheduling Criteria

- **Waiting time**
  - Amount of time a process has been waiting in the ready queue for the processor
- **Response time**
  - For an interactive user, this is the time from submission of a request to until the request begins to be received (first response) .
- **Deadlines**
  - Maximize the number of processes meeting deadlines (Important for real-time jobs)

Scheduling Algorithm Optimization Criteria

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

First – Come – First – Served (FCFS)

- Non-preemptive Scheduling
- At decision point, the oldest process in the Ready queue is selected
- A short process may have to wait a very long time before it can execute
- Favors CPU-bound processes
  - I/O processes have to wait until CPU-bound process completes

First – Come – First – Served (FCFS)

Process	Burst Time
P_1	24
P_2	3
P_3	3

If Arrival order is P_1 , P_2 , P_3 The Gantt Chart for the schedule is:

P_1	P_2	P_3
0	24	27	30

For P1, P2 and P3

- Waiting time {0; 24; 27}, Avg. WT (0 + 24 + 27)/3 = 17
- Turn around time {24; 27; 30},Avg. TAT (24+27+30)/3= 27
- Normalized Turn around time {1; 9; 10}, Avg. NTAT = 6.67

FCFS Scheduler

Suppose that the processes arrive in the order

P_2 , P_3 , P_1

- The Gantt chart for the schedule is:

P_2	P_3	P_1

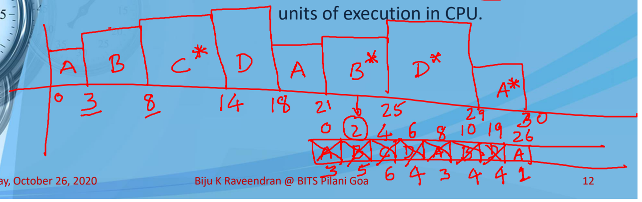
0 3 6 30

- 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

FCFS with I/O

Process	Arrival Time	Execution Time
A	0	7
B	2	9
C	4	6
D	6	8

- Assume process A goes for I/O for 5 units of time after every 3 unit execution in CPU
- Assume B goes for I/O for 2 units after 5 units of execution in CPU
- Process C is a CPU bound process [no I/O]
- Process D goes for I/O for 1 unit after 4 units of execution in CPU.



Operating Systems CS F372

Lect 34: CPU Scheduling

BIJU K RAVEENDRAN

Shortest Job First (SJF) Scheduler

- Associate with each process the length of its next CPU burst. Use 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
- Non-preemptive policy
- Process with shortest expected processing time is selected next
- Short process jumps ahead of longer processes

Example of SJF

Process	Arrival Time	Burst Time
P_1	0.0	6
P_2	1.0	8
P_3	2.0	7
P_4	3.0	3

- SJF scheduling chart

P_1	P_4	P_3	P_2

0 6 9 16 24

- Average waiting time = (0 + 15 + 7 + 3) / 4 = 6.25

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

Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count

80 % → 20 %   20 % → 80 %

- $\alpha = 1$
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

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

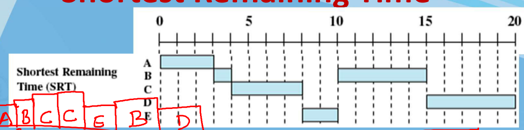
20%

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

Finding Next Burst

- Predictability of longer processes is reduced
- If estimated time for process not correct, the operating system may abort it
- Possibility of starvation for longer processes

Shortest Remaining Time



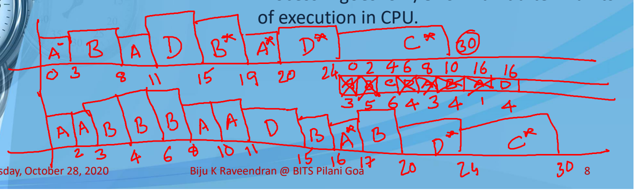
- Preemptive version of Shortest Job First policy
- Must estimate processing time

Process	Arrival Time	Execution Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

SJF and SRTF with I/O

Process	Arr. Time	Execution Time
A	0	7
B	2	9
C	4	6
D	6	8

- Assume process A goes for I/O for 5 units of time after every 3 unit execution in CPU
- Assume B goes for I/O for 2 units after 5 units of execution in CPU
- Process C is a CPU bound process with no I/O
- Process D goes for I/O for 1 unit after 4 units of execution in CPU.

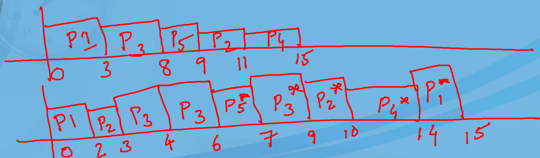


Priority Scheduling

- 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
  - Non-preemptive
- SJF is a priority scheduling where priority is the 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 – Preemptive & non-preemptive Scheduling

Process	Arrival Time	Execution Time	Priority
P1	0	3	5
P2	2	2	3
P3	3	5	2
P4	4	4	4
P5	6	1	1



Operating Systems CS F372

Lect 35: CPU Scheduling

BIJU K RAVEENDRAN

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.
- Uses preemption based on a clock
- 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.

Round Robin

- Clock interrupt is generated at periodic intervals
- When an interrupt occurs, the currently running process is placed in the ready queue
  - Next ready job is selected
- Known as time slicing
- Performance
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high

RR with time Quantum = 4

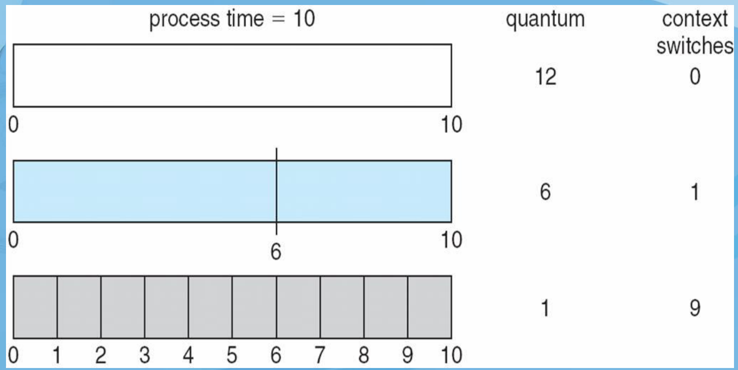
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1
0–4	4–7	7–10	10–14	14–18	18–22	22–26	26–30

- Typically, higher average turnaround than SJF, but better *response*
- *Quantum time should be larger than context switching time (at least 1000 times larger)*

Time Quantum & Context Switching



Priority Scheduling with Round Robin

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

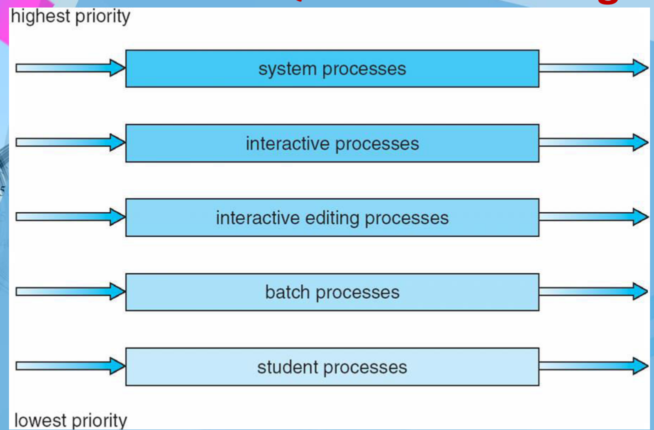
- ❑ Run the process with the highest priority. Processes with the same priority run round-robin
- Gantt Chart with 2 ms time quantum

P_4	P_2	P_3	P_2	P_3	P_2	P_3	P_1	P_5	P_1	P_5
0–7	7–9	9–11	11–13	13–15	15–16	16–20	20–22	22–24	24–26	26–27

Multi-level 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). 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

Multi-level Queue Scheduling



Multi-level 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

Multi-level Feedback Queue

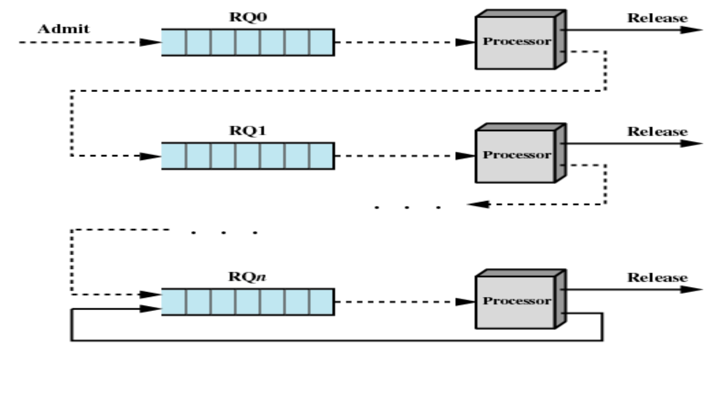
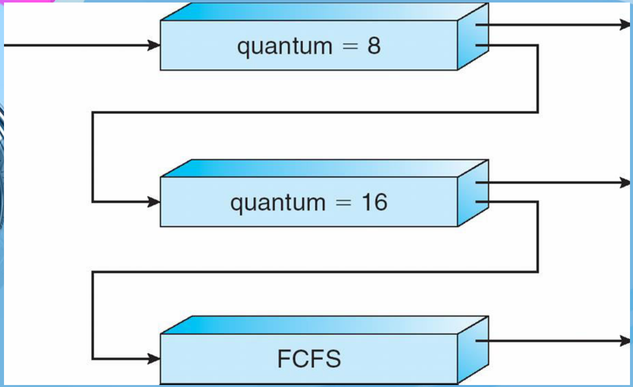


Figure 9.10 Feedback Scheduling

Example of Multi-level 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 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 Q_1.
  – At Q_1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.

Multi-level Feedback Queue



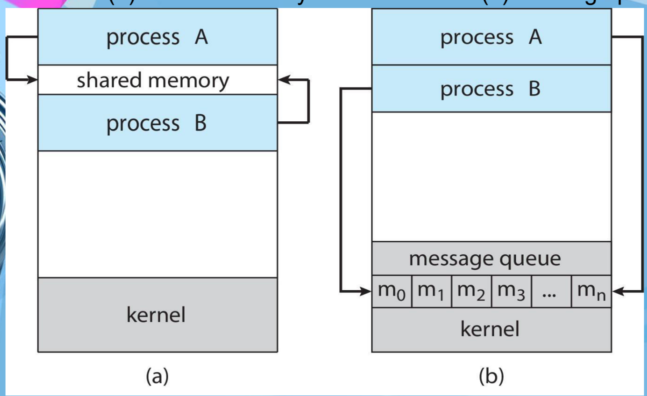
SYNCHRONIZATION

Inter Process Communication

- Processes within a system may be **independent** or **cooperating**
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing, Computation speedup, Modularity, Convenience
- Cooperating processes need **inter process communication** (IPC)
  - Shared memory, Message passing

Communication Models

(a) Shared memory. (b) Message passing.



Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send:** The sender blocks until the message is received
  - **Blocking receive:** The receiver blocks until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking:** The sender sends the message and continue
  - **Non-blocking:** The receiver receives a valid message or null

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

Producer – Consumer Problem

in ← 0, out ← 0

PRODUCER

```
while (1) {
  // Produce item;
  Buffer[in] = item;
  in = in + 1;
}
```

CONSUMER

```
while (1) {
  while (in == out);
  item = Buffer[out];
  out = out +1;
}
```


Bounded Buffer – Shared Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```



```
/* Producer */
while (true) {
  /*  produce an item and put in nextProduced  */
   while (count == BUFFER_SIZE); // do nothing
   buffer [in] = nextProduced;
   in = (in + 1) % BUFFER_SIZE;
   count++;
}

/* Consumer  */
while (true)  {
    while (count == 0); // do nothing
    nextConsumed =  buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
    /*  consume the item in nextConsumed */
}
```

Operating Systems
CS F372

Lect 36: Synchronization

BIJU K RAVEENDRAN

Bounded Buffer – Shared Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int count=0;
```



```
/* Producer */
while (true) {
  /* produce an item and put in nextProduced */
   while (count == BUFFER_SIZE); // do nothing
   buffer [in] = nextProduced;
   in = (in + 1) % BUFFER_SIZE;
   count++;
}
```

count = 0

count ++

Reg1 ← count

Reg1 ← Reg1 + 1

count ← Reg1

-1

```
/* Consumer */
while (true) {
   while (count == 0); // do nothing
   nextConsumed = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   count--;
   /* consume the item in nextConsumed */
}
```

Reg1 ← lw (R1)

addi R1, R1, #1

sw R1, count

1

Producer – Consumer Problem

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Race Condition

- count++ could be implemented as
  - register1 = count
  - register1 = register1 + 1
  - count = register1
- count-- could be implemented as
  - register2 = count
  - register2 = register2 – 1
  - count = register2

Race Condition

Shared var int x =0;

x = -1

~~x = 1~~

-1 → +1

Process #1	**Process #2**
int main ( )	int main ( )
{	{
x = x + 1;	x = x - 1;
return 0;	return 0;
}	}

0

lw R1, $\boxed{x}$ ← CS

addi R1, R1, #1

sw R1, x

x = 1

0

lw R_2, x

addi R_2, R_2, #-1

sw R_2, x

← CS

~~x~~ -1

Race Condition

x = x + 1	**x = x - 1**
Reg #1 ← x (load)	Reg #1 ← x (load)
Reg#1 ← Reg#1 + 1 (compute)	Reg#1 ← Reg#1 - 1 (compute)
x ← Reg #1 (store)	x ← Reg #1 (store)

Inter leaving can happen anywhere in this code

Maximum and Minimum values

- Race condition leads to unpredictable result
- Result can be with in a range
  - Previous example it is –1 to +1.
- Maximum value situation
  - If all x - - operations getting nullify by x + + operation
- Minimum value situation
  - If all x + + operations getting nullify by x - - operation

Race Condition

Shared var int x =0;

Process #1

```
int main ( )
{ int i;
  for(i=0;i<100;i++)
     x = x + 1;
  return 0;
}
```

Process #2

```
int main ( )
{ int i;
  for(i=0;i<100;i++)
     x = x - 1;
  return 0;
}
```

Find Maximum and Minimum value of x

+100
-100
x++
x--
+100
-100

Race Condition

Shared var int x =0;

Process #1

```
int main ( )
{ int i;
  for(i=0;i<100;i++) {
     x = x + 1;
     x = x – 1; }
  return 0;
}
```

Process #2

```
int main ( )
{ int i;
  for(i=0;i<100;i++) {
     x = x + 1;
     x = x - 1; }
  return 0;
}
```

Find Maximum and Minimum value of x

-199
+199

x++
x--

for (i=0; i<5; i++)
{ x++;
x--;
}

-9
+9

Race Condition

- The value of shared variable in execution is dependent on the order of a shared variable or resource.
- Atomic operation
  - The execution of a bunch of operations are atomic if and only if the operation(s) either execute completely or the execution will not take place at all

Critical Section Problem

- Consider system of *n* processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has critical section segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- *Critical section problem* is to design protocol to solve this
- Each process must ask permission to enter critical section in entry section, may follow critical section with exit section, then remainder section