

Operating Systems

Sina Meraji
U of T



Today ...



- Another Synchronization example
- Scheduling

Another Synchronization Example



- Dining philosophers

Higher-level Abstractions for CS's



- Locks
 - Very primitive, minimal semantics
 - Operations: `acquire(lock)`, `release(lock)`
- Semaphores
 - Basic, easy to understand, hard to program with
- Monitors
 - High-level, ideally has language support (Java)
- Messages
 - Simple model for communication & synchronization
 - Direct application to distributed systems



Semaphores

- Semaphores are abstract data types that provide synchronization. They include:
 - An integer variable, accessed only through 2 atomic operations
 - The atomic operation *wait* (also called *P* or *decrement*) - decrement the variable and block until semaphore is free
 - The atomic operation *signal* (also called *V* or *increment*) - increment the variable, unblock a waiting a thread if there are any
 - A queue of waiting threads



Types of Semaphores

- Mutex (or Binary) Semaphore
 - Represents single access to a resource
 - Guarantees mutual exclusion to a critical section
- Counting semaphore
 - Represents a resource with many units available, or a resource that allows certain kinds of unsynchronized concurrent access (e.g., reading)
 - Multiple threads can pass the semaphore
 - Max number of threads is determined by semaphore's initial value, *count*
 - Mutex has $count = 1$, counting has $count = N$

Semaphores



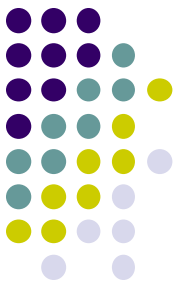
- Integer variable `count` with two atomic operations
 - Operation *wait* (also called *P* or *decrement*)
 - block until `count > 0` then decrement variable

```
wait(semaphore *s) {  
    while (s->count == 0) ;  
    s->count -= 1;  
}
```

- Operation *signal* (also called *V* or *increment*)
 - increment `count`, unblock a waiting thread if any

```
signal(semaphore *s) {  
    s->count += 1;  
    ..... //unblock one waiter  
}
```

- A queue of waiting threads



Using Binary Semaphores

- Use is similar to locks, but semantics are different

Have semaphore, S, associated with acct

```
typedef struct account {  
    double balance;  
    semaphore S;  
} account_t;  
  
Withdraw(account_t *acct, amt) {  
    double bal;  
    wait(acct->S);  
    bal = acct->balance;  
    bal = bal - amt;  
    acct->balance = bal;  
    signal(acct->S);  
    return bal;  
}
```

Three threads execute Withdraw()

```
wait(S);  
bal = acct->balance;  
bal = bal - amt;
```

```
wait(acct->S);
```

```
wait(acct->S);
```

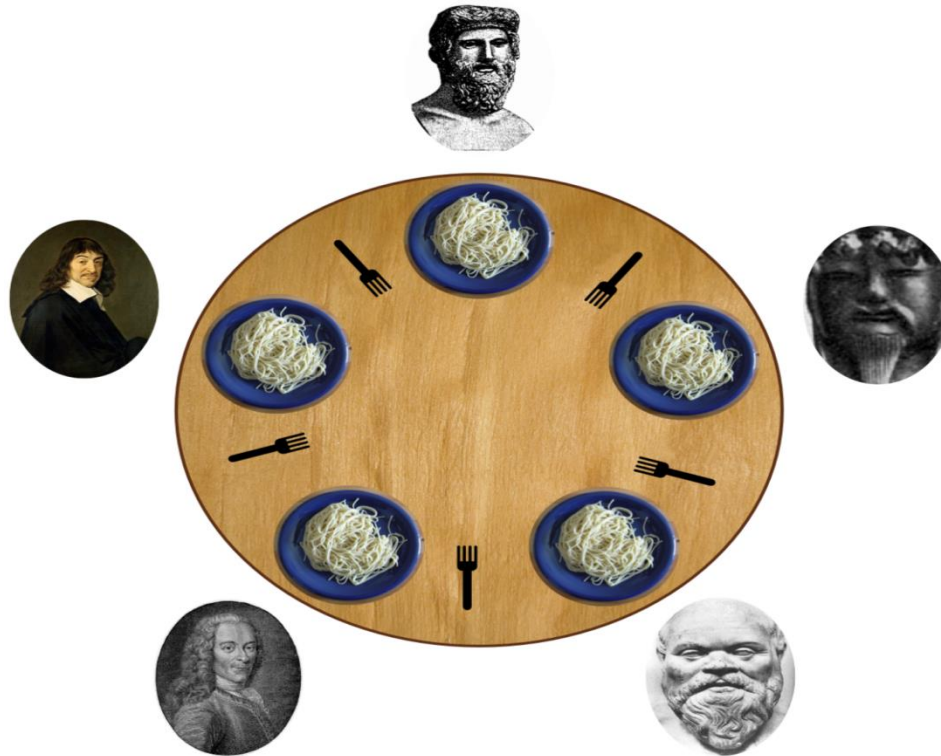
```
acct->balance = bal;  
signal(acct->S);
```

```
...  
signal(acct->S);
```

```
...  
signal(acct->S);
```

It is **undefined** which thread runs after a **signal**

Dining Philosophers



philosopher -> processes
fork -> shared resources

- A philosopher needs two forks to eat.
- Idea for protocol:
 - When philosopher gets hungry grab left fork, then grab right fork.
- Is this a good solution?

Dining Philosophers Problem (1)



```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                             /* philosopher is thinking */
        take_fork(i);                          /* take left fork */
        take_fork((i+1) % N);                  /* take right fork; % is modulo operator */
        eat( );                                /* yum-yum, spaghetti */
        put_fork(i);                           /* put left fork back on the table */
        put_fork((i+1) % N);                   /* put right fork back on the table */
    }
}
```

A nonsolution to the dining philosophers problem.



Dining Philosophers Problem (2)

```
#define N          5                /* number of philosophers */
#define LEFT      (i+N-1)%N        /* number of i's left neighbor */
#define RIGHT     (i+1)%N          /* number of i's right neighbor */
#define THINKING  0                /* philosopher is thinking */
#define HUNGRY    1                /* philosopher is trying to get forks */
#define EATING    2                /* philosopher is eating */
typedef int semaphore;             /* semaphores are a special kind of int */
int state[N];                     /* array to keep track of everyone's state */
semaphore mutex = 1;              /* mutual exclusion for critical regions */
semaphore s[N];                   /* one semaphore per philosopher */
void philosopher(int i)           /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {                /* repeat forever */
        think( );                 /* philosopher is thinking */
        take_forks(i);            /* acquire two forks or block */
        eat( );                   /* yum-yum, spaghetti */
        put_forks(i);             /* put both forks back on table */
    }
}
. . .
```

A solution to the dining philosophers problem.

Dining Philosophers Problem (3)



require mutex semaphore state is synchronized,
this is because adjacent philosophers, in
addition to the philosopher himself, are able to
change this philosopher's state

...

```
void take_forks(int i)                                /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                                     /* enter critical region */
    state[i] = HUNGRY;                                /* record fact that philosopher i is hungry */
    test(i);                                           /* try to acquire 2 forks */
    up(&mutex);                                       /* exit critical region */
    down(&s[i]);                                     /* block if forks were not acquired */
}
...
```

A solution to the dining philosophers problem.

Dining Philosophers Problem (4)

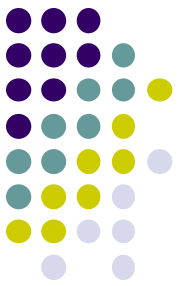


...

```
void put_forks(i)                                /* i: philosopher number, from 0 to N-1 */
{
    thinking and testing -> returning the forks
    down(&mutex);                                /* enter critical region */
    state[i] = THINKING;                         /* philosopher has finished eating */
    test(LEFT);                                  /* see if left neighbor can now eat */
    test(RIGHT);                                 /* see if right neighbor can now eat */
    up(&mutex);                                  /* exit critical region */
}
    signal adjacent philosophers to give
    them semaphore if they are ready to eat

void test(i) /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;                      -> fork at left and right is available
        up(&s[i]);
    }
    increment s[i] so that take_forks() can exit properly, i.e. starts eating..
}
```

A solution to the dining philosophers problem.



Review: The Process Concept

- Process = job / unit of work
- Process = a program in execution
- A process contains all state of program in execution

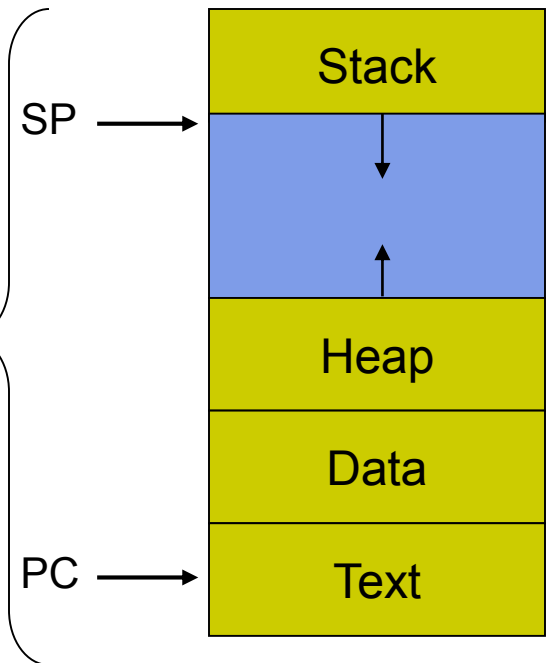
- An address space

- Set of OS resources

- Open files network connections ...

- Set of general-purpose registers with current values

- Accounting info

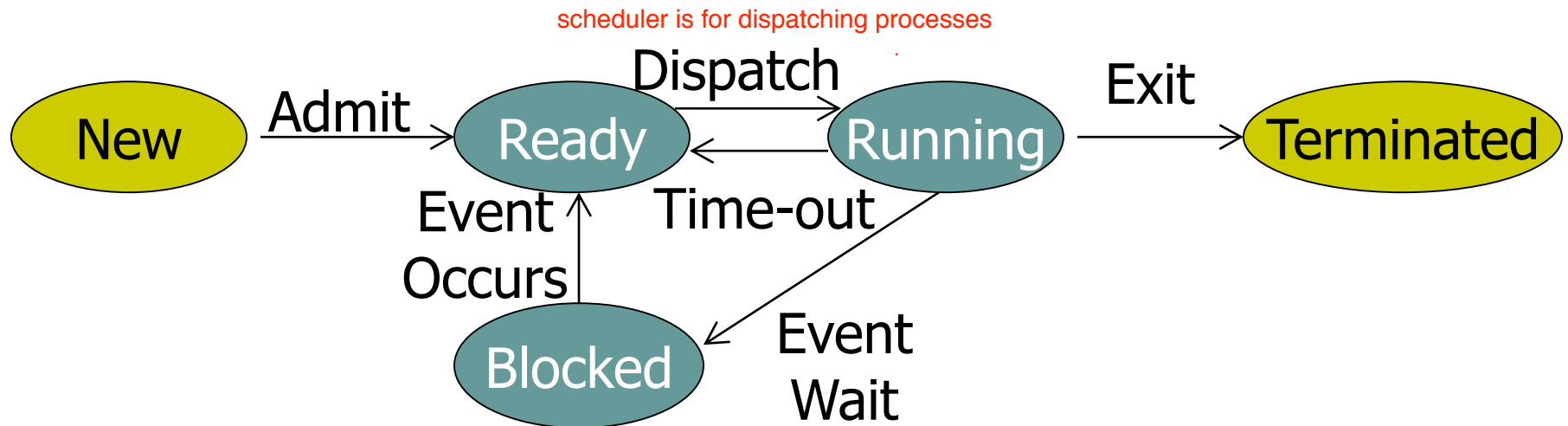


- A process is named by its process ID (PID)



Process states

- The OS manages processes by keeping track of their state BLOCKED, RUNNING, WAITING
 - Different *events* cause changes to a process state, which the OS must record/implement



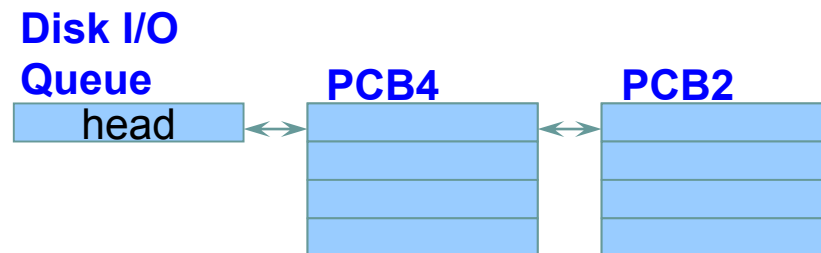
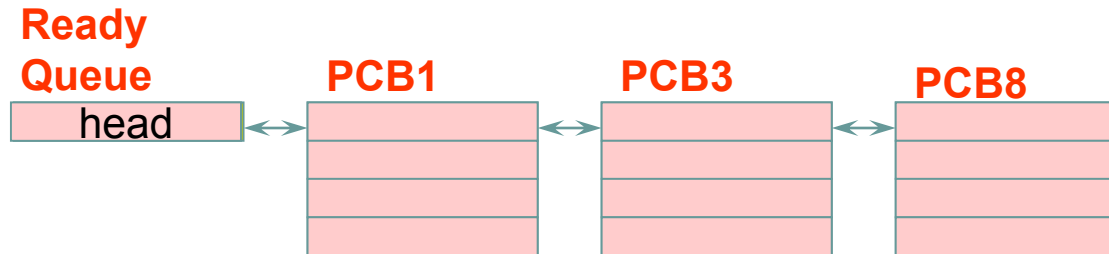


State Queues

How does the OS keep track of processes?

- The OS maintains a collection of queues that represent the state of all processes in the system
- Typically, the OS has one queue for each state
so something like 3 queues
 - Ready, waiting, etc.
- Each PCB is queued on a state queue according to its current state
- As a process changes state, its PCB is unlinked from one queue and linked into another

State Queues



Sleep Queue

•
•
•

- There may be many wait queues, one for each type of wait (disk, console, timer, network, etc.)



Process Scheduling

- Only one process can run at a time on a CPU
- Scheduler decides which process to run
- Goal of CPU scheduling:
 - Give illusion that processes are running concurrently
 - Maximize CPU utilization reduce idle time
- Will talk about CPU scheduling in more detail ...

process life cycle

- + process repeatedly alternate between computation and I/O
- + called CPU bursts and I/O bursts
- + last CPU burst ends with a call to terminate process (`_exit()`)
 - + CPU-bound: very long CPU bursts, infrequent I/O
 - + IO bound: short CPU bursts, frequent I/O bursts
- + so should not wait for I/O bursts since CPU gets idled

Scheduling Goals



- All systems

- **Fairness** - each process receives fair share of CPU
- Avoid starvation
- Policy enforcement - usage policies should be met say some processes should be running on same core as other processes
- Balance - all parts of the system should be busy

- Batch systems

i.e. both CPU and disk should be busy

- **Throughput** - maximize jobs completed per hour
- **Turnaround time** - minimize time between submission and completion
- CPU utilization - keep the CPU busy all the time

Interactive system

+ response time - minimize time between receiving request and starting to produce output

+ proportionality - simple tasks complete quickly

real time system

+ predictability



Types of Scheduling

- **Non-preemptive scheduling**
 - once the CPU has been allocated to a process, it keeps the CPU until it terminates
 - Suitable for batch scheduling
- **Preemptive scheduling** personal computer is preemptive
 - CPU can be taken from a running process and allocated to another
 - Needed in interactive or real-time systems
for system with both CPU bound and IO bound tasks

Scheduling Algorithms: FCFS



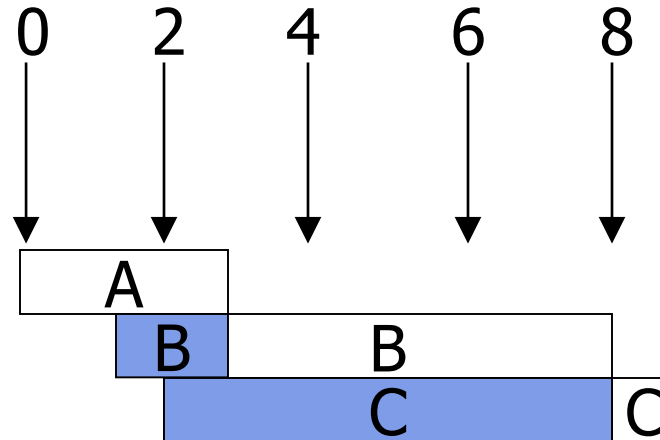
- “First come, first served”
- Non-preemptive
- Choose the process at the head of the FIFO queue of ready processes





FCFS Example

| Process | Arrival Time | Service Time |
|---------|--------------|--------------|
| A | 0 | 3 |
| B | 1 | 5 |
| C | 2 | 1 |



- Note C waits six times as long as it runs!
 - Total wait time is 8, avg. wait is $8/3=2.66$

Problem with FCFS

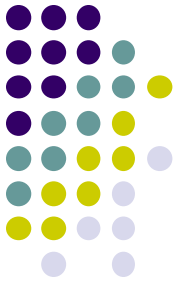


- Average waiting time often quite long
 - **Convoy effect:** all other processes wait for the one big process to release the CPU



What can we do to minimize wait times?

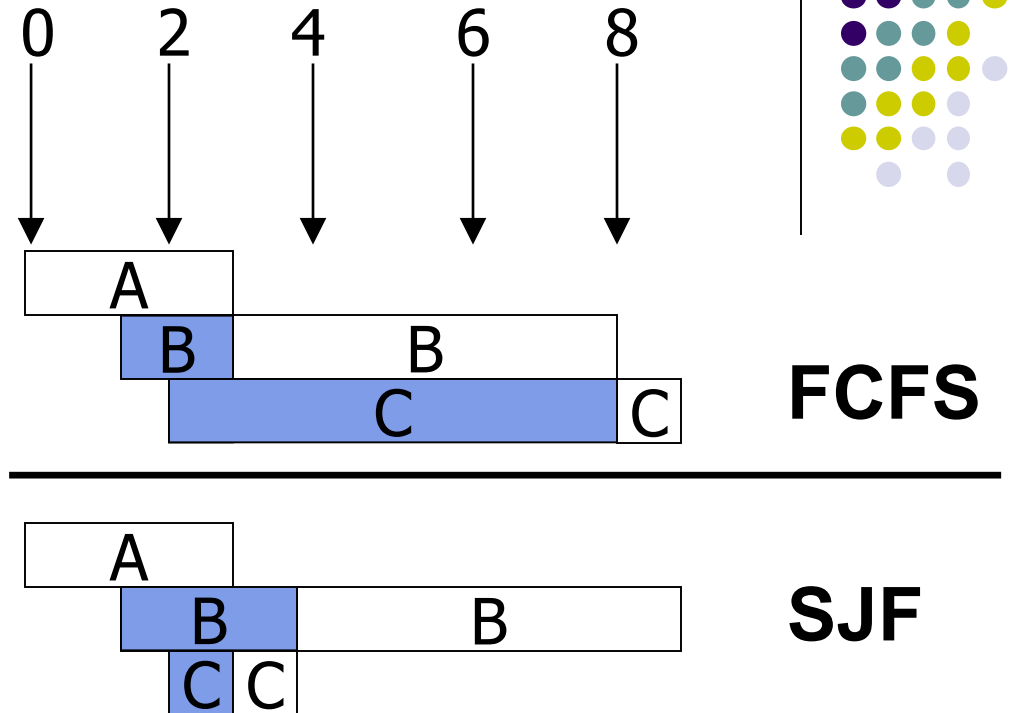
Algorithm: Shortest-Job-First



- Choose the process with the shortest processing time

SJF Example

| Processes | Arrival Time | Service Time |
|-----------|--------------|--------------|
| A | 0 | 3 |
| B | 1 | 5 |
| C | 2 | 1 |



- FCFS: Total wait time is 8, avg. wait is $8/3=2.66$
- SJF: Total wait time is 4, avg. wait is $4/3=1.33$

We cut wait time in half just by scheduling!



Algorithm: Shortest-Job-First

- Choose the process with the shortest processing time
- Provably optimal average wait time
- Problems with SJF?
 - Need to know processing time in advance
 - Programmer estimate which has to be estimated....
 - History statistics
 - Starvation + fairness?
long running processes may experience starvation, unfair amongst a queue of lots of small jobs



So far ...

- SJF has **short wait times**, but is **unfair**
- FCFS is **fair**, but can lead to **long wait times**
 - Short jobs can get stuck behind long jobs

How can we be fair, but avoid the “short jobs getting stuck behind long jobs” problem?

IDEA: Allow preemption

- Don't always run jobs to completion
- Preempt a job that's running too long

Algorithm: Round Robin



- Designed for time-sharing systems
- Pre-emptive
- Ready queue is circular
 - Each process is allowed to run for time quantum q before being preempted and put back on queue
- Choice of quantum (aka time slice) is critical
 - as $q \rightarrow \infty$, RR \rightarrow FCFS; no job ever preempted
 - as $q \rightarrow 0$, RR \rightarrow processor sharing (PS) too many costly context switches
 - we want q to be large w.r.t. the context switch time



Priority Scheduling

- A priority, p , is associated with each process
- Highest priority job is selected from Ready queue
maybe different queue for different priorities
- Can be pre-emptive or non-preemptive
- Enforcing this policy is tricky processes may have shared resources...
 - A low priority task may never get to run (*starvation*)
 - A low priority task may prevent a high priority task from making progress by holding a resource (*priority inversion*)



Priority Inversion Example

- Mars Rover *Pathfinder* bug
 - In the press:
 - “software glitches”,
 - “computer was doing too many things at once”
 - + low priority task gets a lock, preempted by high priority task,
 - + high priority task is blocked since lock unavailable
 - + medium priority arrives, executed before low priority task (since higher priority) and executed before high priority task (priority inversion)

“bus management”
HIGH priority



“data gathering”
LOW priority



“communications task”
(does not use bus)
MED priority



Shared memory
(exclusive use)



Priority Inversion Example

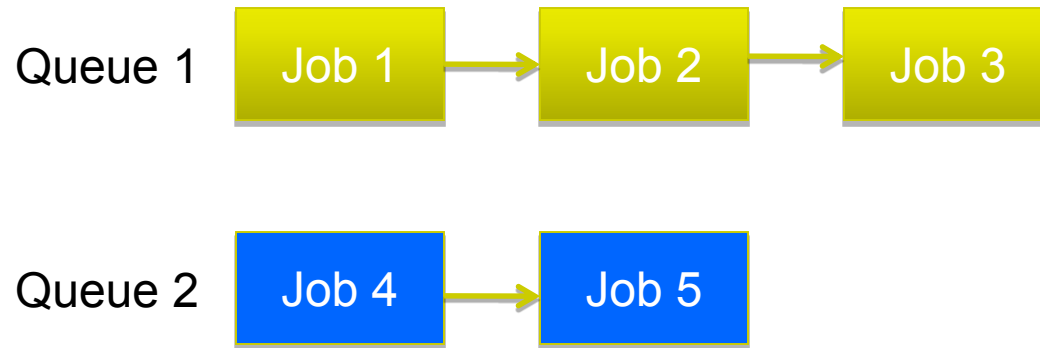
- Mars Rover *Pathfinder* bug
 - Shared “information bus” – essentially shared memory area
 - Mutual exclusion provided by lock on info bus
 - High priority “bus management” task moves data in/out of information bus
 - Low priority “data gathering” task writes data to the information bus
 - Medium priority, compute-bound “communications” task that does not use the information bus
 - See the problem?
 - Data gathering task locks bus and is preempted by higher priority bus management task, which blocks on the lock. If communications task becomes runnable, data gathering task can't complete and release the lock so high priority task stays blocked.



What do real systems do?

- Combination of
 - multiple queue, each queue with different types (priority, interactive?) of tasks, as well as different scheduling algorithms (different quantum q different algo)
 - Multi-level queue scheduling
 - Typically with RR and priorities
 - Feedback scheduling
- What does that mean ...

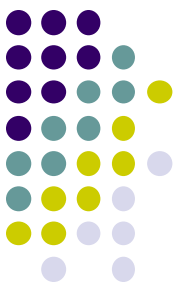
Multi-Level Queue Scheduling



- Have two scheduling policies
 - First decides which queue to serve
 - E.g. based on queue priorities
 - Second decides which job within a queue to choose
 - E.g. FCFS or RR

what queue to pick next and what job
in that queue to pick from (algo)

Multiple Queues with Priority Scheduling



- Highest priority gets one quantum, second highest gets 2.....
 - If highest finishes during quantum, great. Otherwise bump it to second highest priority and so on into the night
- Consequently, shortest (high priority) jobs get out of town first
- They announce themselves-no previous knowledge assumed!



Feedback Scheduling

- Motivation:
 - Want to give priority to shorter jobs
 - Want to give priority to IO bound jobs
 - Want to give priority to interactive jobs
 - Want to ...
- But don't know beforehand whether a job is short or long and whether it's IO bound or CPU bound ...



Feedback Scheduling

change priority based on observation

- Adjust criteria for choosing a particular process based on past history
 - Combine with MLQ
 - Can prefer processes that do not use full quantum
 - Can change priority of processes based on age
 - Can change priority of processes based on CPU consumed so far
 - Can boost priority following a user-input event
- Move processes between queues

Linux 2.6 CPU scheduling

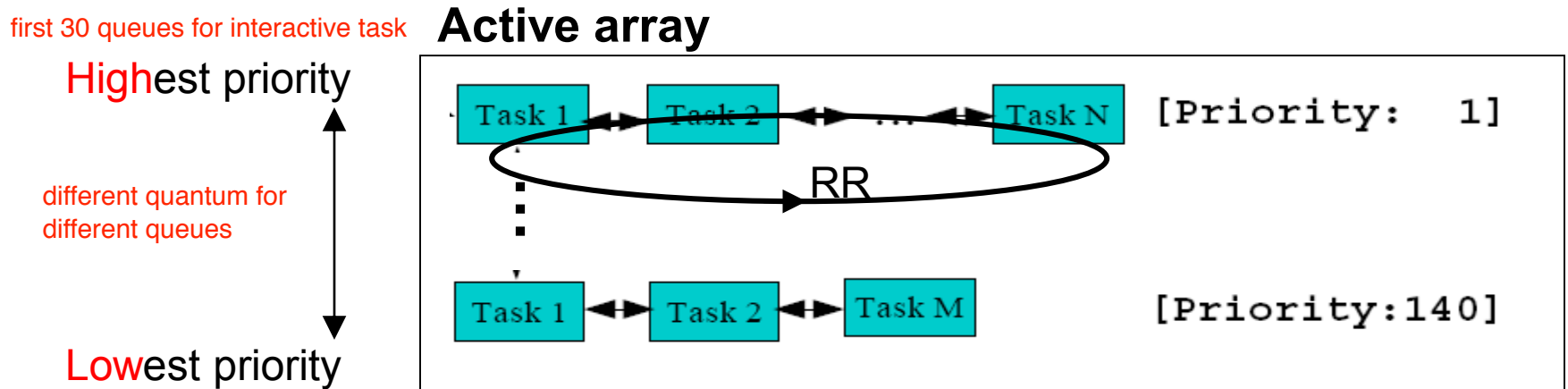


- Combination of
 - Multilevel queues
 - With priorities and RR
 - Feedback scheduling
- Distinguishes 3 classes
 - Realtime FIFO processes
 - Realtime RR processes
 - Timesharing processes
- Our discussion focuses on timesharing processes

Linux 2.6 CPU scheduling



- Always run the task in active array with highest priority
- If there are procs with same priority, do RR between them
 - Switch after **granularity** time units



How do you avoid **starvation** of low priority processes?

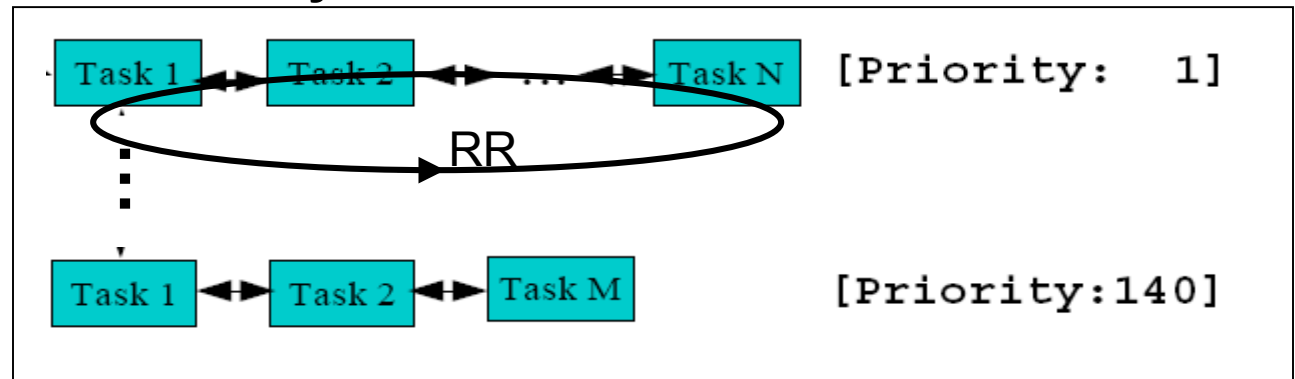
Linux 2.6 CPU scheduling



- Always run the task in active array with highest priority
- If there are procs with same priority, do RR between them
 - Switch after **granularity** time units
- If **timeslice** expires move to “Expired array”

Active array

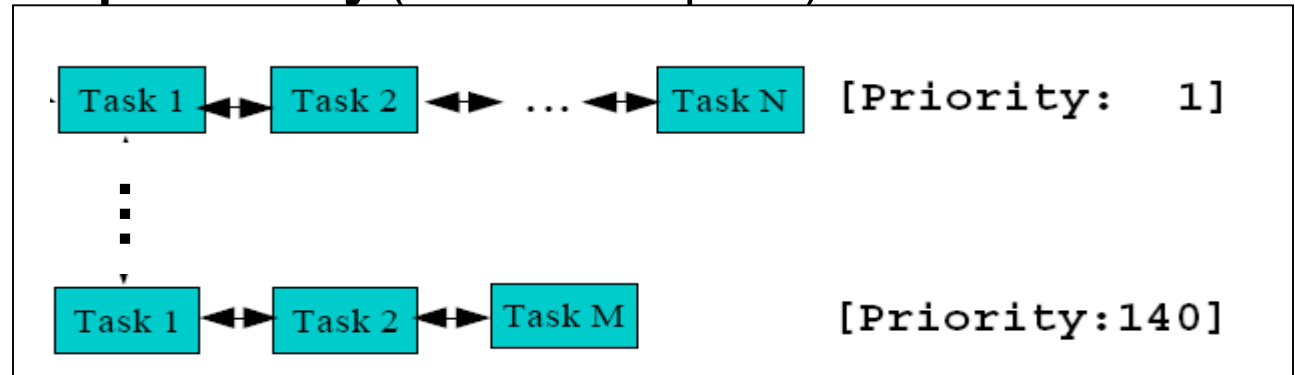
Highest priority



Lowest priority

if process did not finish, put task from active to expired array. this repeats until timeslice expires, now expired array turns to active array, placement of task may change based on previous performance...

Expired array (timeslice expired)





Linux 2.6 CPU scheduling

- How to determine priority of process?
 - Some processes are more important than others
 - Assign ^{user-determined} **static priorities** [-20, 19]
 - Done through **nice** system call
 - Interactive & IO bound processes should be favored
 - Give bonus/penalty on top of static priority
 - **priority = static_priority + ^{dynamically determined} bonus**
- How to identify interactive / IO bound processes?
 - Maintain **sleep_avg** (time process is sleeping vs CPU time)

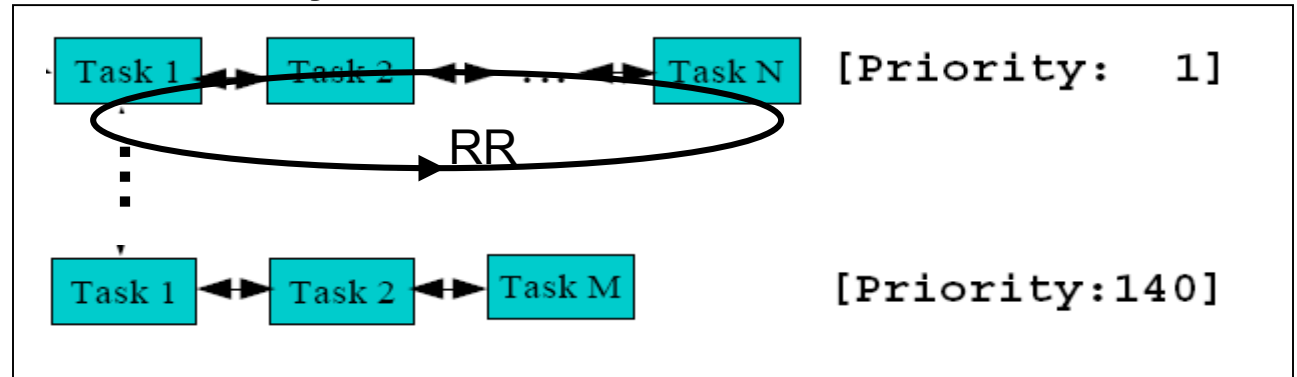
Linux 2.6 CPU scheduling



- Always run the task in active array with highest priority
- If there are procs with same priority, do RR between them
 - Switch after **granularity** time units
- If **timeslice** expires move to “Expired array”

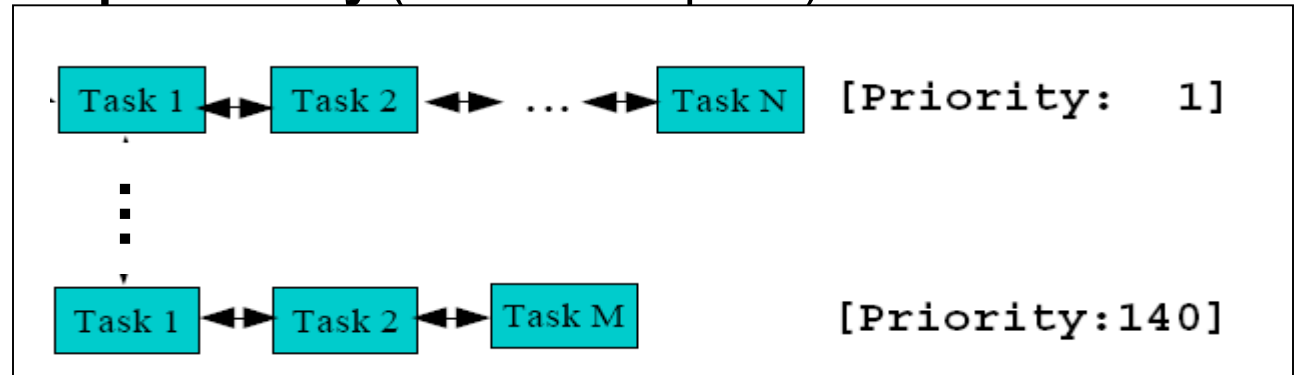
Active array

Highest priority



Lowest priority

Expired array (timeslice expired)





Linux 2.6 CPU scheduling

- How to set `timeslice` and `granularity`?
 - `Timeslice` depends on static priority
 - Between 5ms and 800ms
 - `Granularity` depends on dynamic priority
- Many of the details keep changing ...

New topic next week:

- Memory management!

