0.0.1 Semaphores

- Dijkstra (1965) introduced the concept of a semaphore
- A semaphore is an integer variable that is accessed through two standard atomic operations: wait (a spinlock, i.e. stops blocking and decrements thesemaphore) and signal (i.e. the semaphore counts the signals it receives)
- Semaphores are variables that are used to signal the status of shared resources to processes (a semaphore could have the value of 0, indicating that no wakeups are saved, or some positive value if one or more wakeups are pending)
- How does that work?
  - If a resource is not available, the corresponding semaphore blocks any process waiting for the resource
  - Blocked processes are put into a process queue maintained by the semaphore (avoids busy waiting!)
  - When a process releases a resource, it signals this by means of the semaphore
  - Signalling resumes a blocked process if there is any
  - Wait and signal operations cannot be interrupted
  - Complex coordination can be specified by multiple semaphores
- the *down* operation on a semaphore
  - checks to see if the value is greater than 0
  - if so, it decrements the value and continues
  - if the value is 0, the process is put to *sleep* without the completing the *down* for the moment
  - all is done as a single, indivisible atomic action
    - checking the value
    - changing it
    - possibly going to sleep
  - it is guaranteed that once a semaphore operation has started, no other process can access the semaphore until the operation has completed

  - synchronization and no race condition

- the *up* operation on a semaphore
  - increments the value of the semaphore
  - if one or more processes were sleeping on that semaphore, unable to complete an earlier *down* operation, one of them is chosen by the system and allowed to complete its *down*
  - the semaphore will be 0. but there will be one fewer process sleeping on it
  - indivisible process; incrementing the semaphore and waking up one process

- Solving the producer-consumer problem using semaphores (see Fig. 1)
  - the solution uses three semaphores;
    * one called **full** for counting the number of slots that are full
    * one called **empty** for counting the number of slots that are empty
    * one called **mutex** to make sure the producer and the consumer do not access the buffer at the same time. **mutex** is initially 1 (**binary semaphore**)
    * if each process does a *down* just before entering its CR and an *up* just after leaving it, the mutual exclusion is guaranteed.

- Possible uses of semaphores;
  - Mutual exclusion, initialize the semaphore to one
  - Synchronization of cooperating processes (signaling), initialize the semaphore to zero
  - Managing multiple instances of a resource, initialize the semaphore to the number of instances

- Type of semaphores;
  - **binary** is a semaphore with an integer value of 0 and 1.
  - **counting** is a semaphore with an integer value ranging between 0 and an arbitrarily large number. Its initial value might represent the number of units of the critical resources that are available. This form is also known as a general semaphore.

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE is the constant 1 */
          item = produce_item( );               /* generate something to put in buffer */
          down(&empty);                         /* decrement empty count */
          down(&mutex);                         /* enter critical region */
          insert_item(item);                    /* put new item in buffer */
          up(&mutex);                           /* leave critical region */
          up(&full);                            /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* infinite loop */
          down(&full);                          /* decrement full count */
          down(&mutex);                         /* enter critical region */
          item = remove_item( );                /* take item from buffer */
          up(&mutex);                           /* leave critical region */
          up(&empty);                           /* increment count of empty slots */
          consume_item(item);                   /* do something with the item */
     }
}
```

Figure 1: The producer-consumer problem using semaphore

0.0.2 Monitors

- Semaphores are useful and powerful
- But, they require programmer to think of every timing issue; easy to miss something, difficult to debug
- Let the compiler handle the details
- Monitors are a high level language construct for dealing with synchronization
  - similar to classes in Java
  - a monitor has fields and methods
- A monitor is a software module implementing mutual exclusion
- Monitors are easier to program than semaphores
  - programmer only has to say what to protect
  - compiler actually does the protection (compiler will use semaphores to do protection)
- Natively supported by a number of programming languages: Java
  - Resources or critical sections can be protected using the keyword: **synchronized** keyword
  - **synchronized** can be applied to a method: entire method is a critical section
- Chief characteristics (see Fig. 2):
  - Local data variables are accessible only by the monitor (not externally)
  - Process enters monitor by invoking one of its procedures, but cannot directly access the monitor's internal datastructures
  - Only one process may be executing in the monitor at a time (mutual exclusion)
  - Only methods inside monitor can access fields
  - At most one thread can be active inside monitor at any one time
- Main problem: provides less control

- Allow process to wait within the monitor with **condition** variable, condition x,y;
- can only be used with operations **wait** and **signal** (notify() in Java);
  - operation **wait(x)**; means that the process invoking this operation is suspended until another process invokes **signal(x)**;
  - operation **signal(x)**; resumes exactly one process suspended
- condition variables are not counters, they do not accumulate signals for later use the way the semaphores do. Thus if a condition variable is signaled with no waiting on it, the signal is lost
- This solution is deadlock free
- In Fig. 3, the solution for the producer-consumer problem with a monitor is given
- The class **our_monitor** contains the buffer, the administration variables and two synchronized methods
- when the producer is active in *insert*, it knows for sure that the consumer can not be active inside *remove*
- making it safe to update the variables and buffer without fear of race conditions

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

        procedure consumer( );
        .
        .
        .
        end;
end monitor;
```

Figure 2: A monitor

```
monitor ProducerConsumer
        condition full, empty;
        integer count;
        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;
        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count - 1;
                if count = N - 1 then signal(full)
        end;
        count := 0;
end monitor;

procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;
procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

Figure 3: The producer-consumer problem with a monitor.

0.1 Classical IPC Problems

0.1.1 The Dining Philosophers Problem (see Fig. 4)

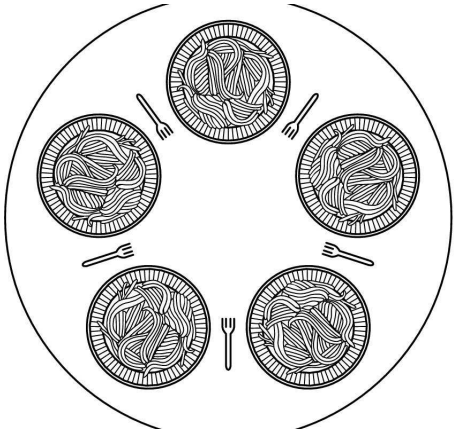


Figure 4: Lunch time in the Philosophy Department.

- Five philosophers are seated around a circular table
- A philosopher needs two forks to eat
- The life of a philosopher consists of alternate periods of eating and thinking
- Write a program for each philosopher that does what it is supposed to do and never gets stuck
- one attempt is to use a binary semaphore (**think**)
  - before starting to acquire forks, a philosopher would do a **down** on *mutex*
  - after replacing the forks, he would **up** on *mutex*
  - bug: only one philosopher can be eating at any instant
- the solution presented in Fig. 5 uses an array, *state*, to keep track of whether a philosopher is eating, thinking, or hungry (trying to acquire forks)
- A philosopher may move only into eating state if neither neighbor is eating

```
#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 */
    }
}

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 */
}

void put_forks(i)                       /* i: philosopher number, from 0 to N−1 */
{
    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 */
}

void test(i)                            /* i: philosopher number, from 0 to N−1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Figure 5: A solution to the dining philosophers problem.

- The solution is deadlock–free and allows the maximum parallelism for any number of philosophers

0.1.2 The Readers and Writers Problem (see Fig. 6)

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to 'rc' */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
     while (TRUE) {                     /* repeat forever */
          down(&mutex);                 /* get exclusive access to 'rc' */
          rc = rc + 1;                  /* one reader more now */
          if (rc == 1) down(&db);       /* if this is the first reader ... */
          up(&mutex);                   /* release exclusive access to 'rc' */
          read_data_base( );            /* access the data */
          down(&mutex);                 /* get exclusive access to 'rc' */
          rc = rc – 1;                  /* one reader fewer now */
          if (rc == 0) up(&db);         /* if this is the last reader ... */
          up(&mutex);                   /* release exclusive access to 'rc' */
          use_data_read( );             /* noncritical region */
     }
}


void writer(void)
{
     while (TRUE) {                     /* repeat forever */
          think_up_data( );             /* noncritical region */
          down(&db);                    /* get exclusive access */
          write_data_base( );           /* update the data */
          up(&db);                      /* release exclusive access */
     }
}
```

Figure 6: A solution to the readers and writers problem.

- Models access to a database; many competing processes wishing to read and write
- It is acceptable to have multiple processes reading the database at the same time, but if one process is updating (writing) the database, no other process may have access to the database, not even readers
- Write a program for the readres and writers

- the solution presented in Fig. 6, the first reader to get access to the database does a **down** on the semaphore *db*
- Subsequent readers increment a counter, *rc*
- As readers leave, they decrement the counter and the last one does an **up** on the semaphore, allowing a blocked writer
- bug:
  - As long as at least one reader is still active, subsequent readers is admitted
  - As a consequence of this strategy, as long as there is a steady supply of readers. they will all get in as soon as they arrive
  - The writer will be kept suspended until no reader is present
- The solution is that when a reader arrives and a write is waiting, the reader is suspended behind the writer instead of being admitted immediately (less concurrency, lower performance)

0.1.3 The Sleeping Barber Problem (see Fig. 7)

- This problem is similar to various queueing situations
- The problem is to program the barber and the customers without getting into race conditions
  - Solution uses three semaphores:
    - *customers*; counts the waiting customers
    - *barbers*; the number of barbers (0 or 1)
    - *mutex*; used for mutual exclusion
    - also need a variable *waiting*; also counts the waiting customers (reason; no way to read the current value of semaphore)
  - The barber executes the procedure *barber*, causing him to block on the semaphore *customers* (initially 0)
  - The barber then goes to sleep
  - When a customer arrives, he executes *customer*, starting by acquiring *mutex* to enter a critical region
  - if another customer enters, shortly thereafter, the second one will not be able to do anything until the first one has released *mutex*

```
#define CHAIRS 5                    /* # chairs for waiting customers */

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void)
{
     while (TRUE) {
          down(&customers);         /* go to sleep if # of customers is 0 */
          down(&mutex);             /* acquire access to 'waiting' */
          waiting = waiting – 1;    /* decrement count of waiting customers */
          up(&barbers);             /* one barber is now ready to cut hair */
          up(&mutex);               /* release 'waiting' */
          cut_hair( );              /* cut hair (outside critical region) */
     }
}


void customer(void)
{
     down(&mutex);                  /* enter critical region */
     if (waiting < CHAIRS) {        /* if there are no free chairs, leave */
          waiting = waiting + 1;    /* increment count of waiting customers */
          up(&customers);           /* wake up barber if necessary */
          up(&mutex);               /* release access to 'waiting' */
          down(&barbers);           /* go to sleep if # of free barbers is 0 */
          get_haircut( );           /* be seated and be serviced */
     } else {
          up(&mutex);               /* shop is full; do not wait */
     }
}
```

Figure 7: A solution to the sleeping barber problem.

- The customer then checks to see if the number of waiting customers is less than the number of chairs
- if not, he releases *mutex* and leaves without a haircut
- if there is an available chair, the customer increments the integer variable, *waiting*
- Then he does an **up** on the semaphore *customers*
- When the customer releases *mutex*, the barber begins the haircut

0.2 Scheduling

- In multiprogramming systems, where there is more than one process runnable (i.e., ready), the operating system must decide which one to run next
- The decision is made by the part of the operating system called the *scheduler*, using a *scheduling algorithm* or *scheduling discipline.*

0.3 Introduction to Scheduling

- **In the beginning**–there was no need for scheduling, since the users of computers lined up in front of the computer room or gave their job to an operator
- **Batch processing**–the jobs were executed in first come first served manner
- **Multiprogramming**–life became complicated!
- The scheduler is concerned with deciding *policy*, not providing a *mechanism*
- The dispatcher is the mechanism
- Dispatcher
  - Low-level mechanism
  - Responsibility: Context-switch
    - Save execution state of old process in PCB
    - Load execution state of new process from PCB to registers
    - Change scheduling state of process (**running, ready,** or **blocked**)
    - Switch from kernel to user mode
    - Jump to instruction in user process
- Scheduler
  - Higher-level policy
  - Responsibility: Deciding which process to run
- Scheduling refers to a set of policies and mechanisms to control the order of work to be performed by a computer system.

- Of all the resources of a computer system that are scheduled before use, the CPU is the far most important.
- But, other criteria may be important too (e.g.,memory)
- Multiprogramming is the (efficient) scheduling of the CPU
- Metrics
  - Execution time: T_s
  - Waiting time: time a thread waits for execution: T_w
  - Turnaround time: time a thread spends in the system (waiting plus execution time): $T_s + T_w = T_r$
  - Normalized turnaround time: T_r/T_s
- Process Behavior
  - The basic idea is to keep the CPU busy as much as possible by executing a (user) process until it must wait for an event and then switch to another process
  - Processes alternate between consuming CPU cycles (*CPU-burst*) and performing I/O (*I/O-burst*)
- Categories of Scheduling Algorithms (See Fig. 8)
  - In general, scheduling policies may be *preemptive* or *non- preemptive*
  - In a non-preemptive pure multiprogramming system, the short-term scheduler lets the current process run until it blocks, waiting for an event or a resource, or it terminates. First-Come-First-Served (FCFS), Shortest Job first (SJF). Good for "background" batch jobs.
  - Preemptive policies, on the other hand, force the currently active process to release the CPU on certain events, such as a clock interrupt, some I/O interrupts, or a system call. Round-Robin (RR), Priority Scheduling. Good for "foreground" interactive jobs
- Scheduling Algorithm Goals
  - A typical scheduler is designed to select one or more primary performance criteria and rank them in order of importance

All systems

Fairness - giving each process a fair share of the CPU
Policy enforcement - seeing that stated policy is carried out
Balance - keeping all parts of the system busy

Batch systems

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

Interactive systems

Response time - respond to requests quickly
Proportionality - meet users' expectations

Real-time systems

Meeting deadlines - avoid losing data
Predictability - avoid quality degradation in multimedia systems

Figure 8: Some goals of the scheduling algorithm under different circumstances.

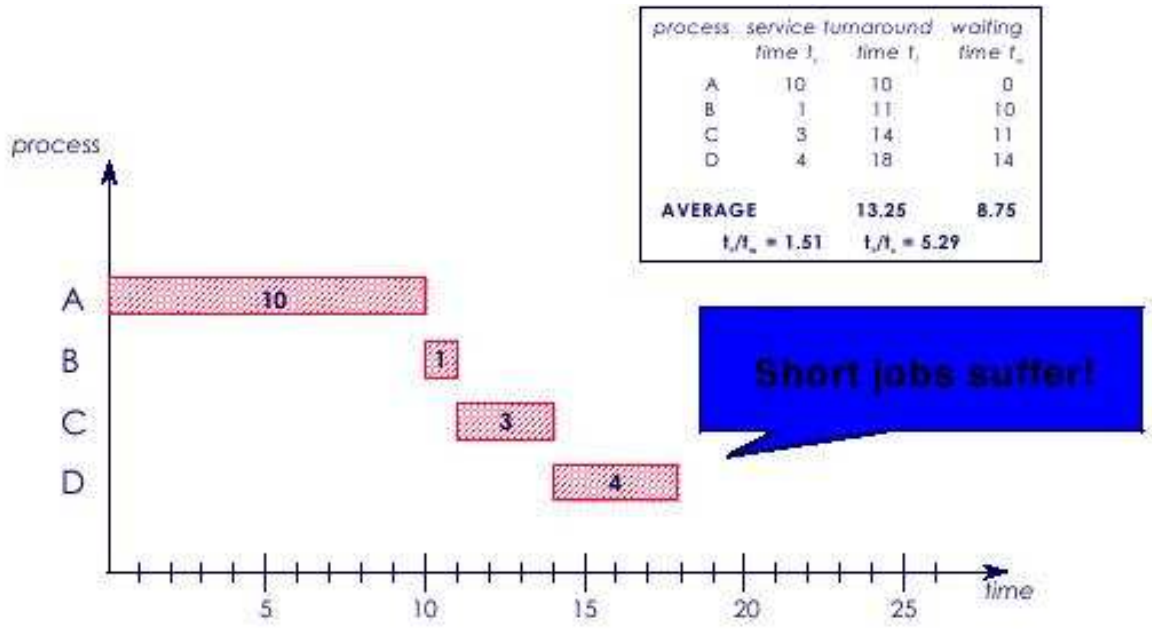
- One problem in selecting a set of performance criteria is that they often conflict with each other
- For example, increased processor utilization is usually achieved by increasing the number of active processes, but then response time decreases
- So, the design of a scheduler usually involves a careful balance of all requirements and constraints
- The following is only a small subset of possible characteristics:*I/O throughput, CPU utilization, response time (batch or interactive), urgency of fast response, priority, maximum time allowed, total time required.*
- Maximize:
  - CPU utilization
  - throughput (number of tasks completed per time unit, also called bandwidth)
- Minimize:
  - Turnaround time (submission to completion, also called latency)
  - Waiting time (sum of time spent in Ready-queue)

    - Response time (time from start of request to production of first response, not full time for output)
  - Fairness:
    - every task should be handled eventually (no starvation)
    - tasks with similar characteristics should be treated equally
  - different type of systems have different priorities!

0.4 Scheduling in Batch Systems

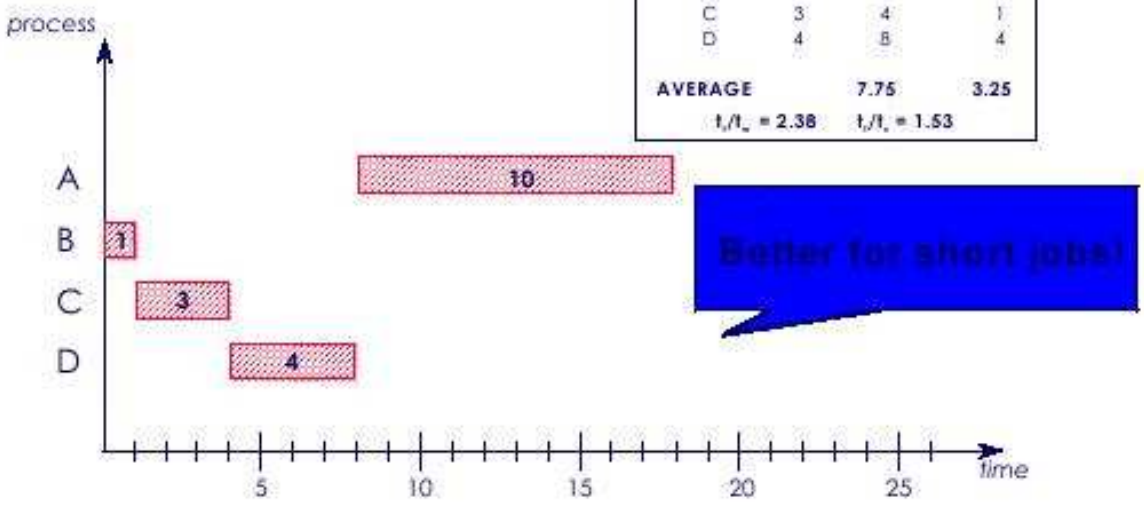
- First-Come First Served (FCFS) (See Fig. 9)
  - FCFS, also known as First-In-First-Out (FIFO), is the simplest scheduling policy
  - Arriving jobs are inserted into the tail of the ready queue and the process to be executed next is removed from the head (front) of the queue
  - FCFS performs better for long jobs
  - Relative importance of jobs measured only by arrival time (poor choice)
  - A long CPU-bound job may take the CPU and may force shorter (or I/O-bound) jobs to wait prolonged periods
  - This in turn may lead to a lengthy queue of ready jobs, and thence to the "convoy effect

- Shortest Job First (SJF)(See Fig. 10)
  - SJF policy selects the job with the shortest (expected) processing time first
  - Shorter jobs are always executed before long jobs
  - One major difficulty with SJF is the need to know or estimate the processing time of each job (can only predict the future!)
  - Also, long running jobs may starve for the CPU when there is a steady supply of short jobs
  - SJF is optimal Â minimum average waiting time for given set of processes
  - nonpreemptive Â once CPU given to process, can't be preempted until completes CPU burst



process	service time t_s	turnaround time t_t	waiting time t_w
A	10	10	0
B	1	11	10
C	3	14	11
D	4	18	14
AVERAGE		13.25	8.75
		t_t/t_s = 1.51	t_t/t_s = 5.29

Figure 9: An example to First-Come First Served.



process	service time t_s	turnaround time t_t	waiting time t_w
A	10	18	8
B	1	1	0
C	3	4	1
D	4	8	4
AVERAGE		7.75	3.25
		t_t/t_s = 2.38	t_t/t_s = 1.53

Figure 10: An example to Shortest Job First.