Systems:

- *Continuous systems*
  State changes continuously in time (e.g., in chemical applications)
- *Discrete systems*
  State is observed at fixed regular time points (e.g., periodic review inventory system)
- *Discrete-event systems*
  The system is completely determined by random event times $t_1, t_2, \ldots$ and by the changes in state taking place at these moments (e.g., production line, queueing system)

Time advance:

- Look at regular time points $0, \Delta, 2\Delta, \ldots$ (*synchronous* simulation); in continuous systems it may be necessary to take Δ very small
- Jump from one event to the next and describe the changes in state at these moments (*asynchronous* simulation)

We will concentrate on asynchronuous simulation of discrete-event systems

Terms often used:

- **System**
  Collection of objects interacting through time (e.g. production system)
- **Model**
  Mathematical representation of a system (e.g., queueing or fluid model)
- **Entity**
  An object in a system (e.g., jobs, machines)
- **Attribute**
  Property of an entity (e.g., arrival time of a job)
- **Linked list**
  Collection of *records* chained together

- **Event**
  Change in state of a system
- **Event notice**
  Record describing when event takes place
- **Process**
  Collection of events ordered in time
- **Future-event set**
  Linked list of event notices ordered by time (FES)
- **Timing routine**
  Procedure maintaining FES and advancing simulated time

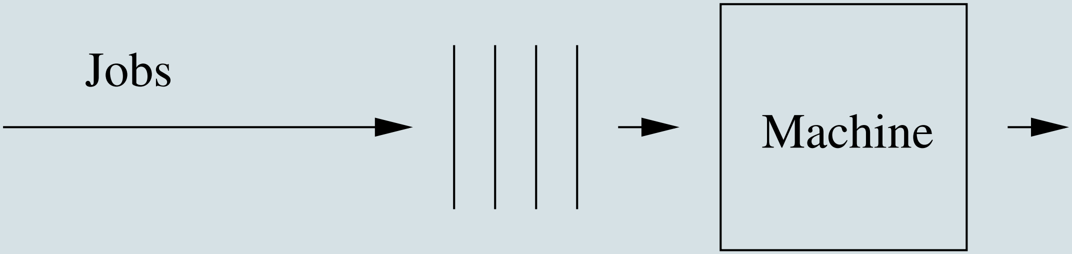
Basic approaches for constructing a discrete-event simulation model:

- *Event-scheduling approach*
  Focuses on events, i.e., the moments in time when state changes occur
- *Process-interaction approach*
  Focuses on processes, i.e., the flow of each entity through the system

In general-purpose languages one mostly uses the event-scheduling approach; simulation languages (e.g., χ) use the process-interaction approach

Event-scheduling approach

Example: Single-stage production system



A single machine processes jobs in order of arrival. The interarrival times and processing times are exponential with parameters λ and μ (with $\lambda < \mu$).

- What is the mean waiting time?
- What is the mean queue length?
- What is the mean length of a busy period?
- How does the performance change if we speed up the machine?

Discrete simulation:

A_n the interarrival time between job n and $n+1$

B_n the processing time of job n

W_n the waiting time of job n

Then (Lindley's equation):

$$W_{n+1} = \max(W_n + B_n - A_n, 0)$$

Initialization

```
n = 0 {job number}
w = 0 {waiting time of job n
       we assume that initially the system is empty}
sum_w = 0 {sum of all waiting times upto job n}
```

Main program

```
while (n < N)
do
    a = interarrival_time
    b = service_time
    w = max(w + b - a, 0)
    sum_w = sum_w + w
    n = n + 1
end
```

Output

```
Mean waiting time = sum_w / N
```

Discrete-event simulation:

Entity	**Attribute**
Job	Arrival time
Machine	Status (idle or busy)

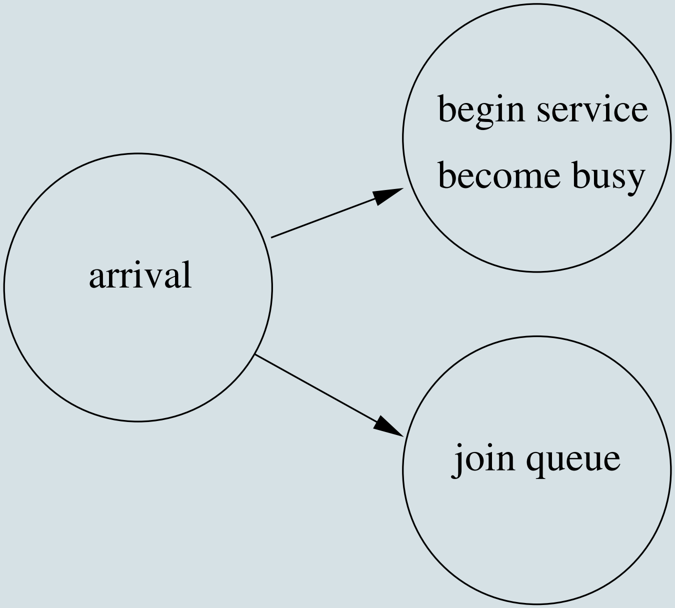
Job is a *temporary* entity
Machine is a *permanent* entity

Elementary events

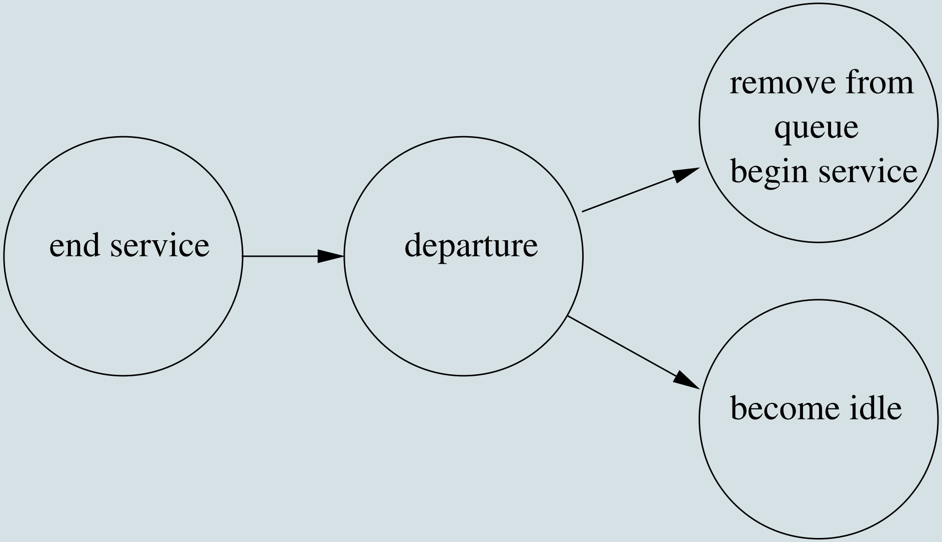
Job:	**Machine:**
arrival	remove from queue
departure	become busy
begin service	become idle
end service	
join queue	

Compound events

Arrival



Departure


end service
departure
remove from
queue
begin service
become idle

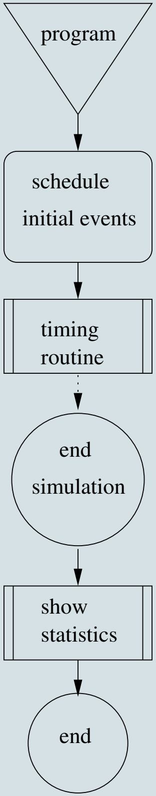
State of the system at time t:

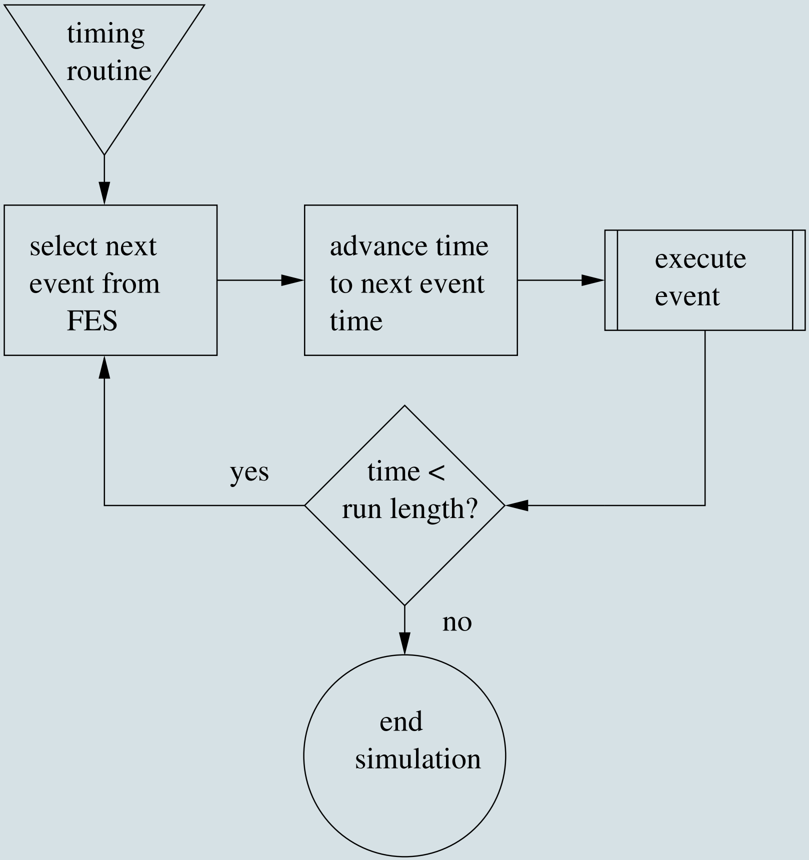
- status of the machine ($i = 0, 1$)
- number of jobs in the queue ($n = 0, 1, 2, \ldots$)
- remaining interarrival time ($a \geq 0$)
- remaining service time ($b \geq 0$)

Then the remaining time until the next event is given by

$$\min(a, b)$$

Prototypical event-scheduling approach:



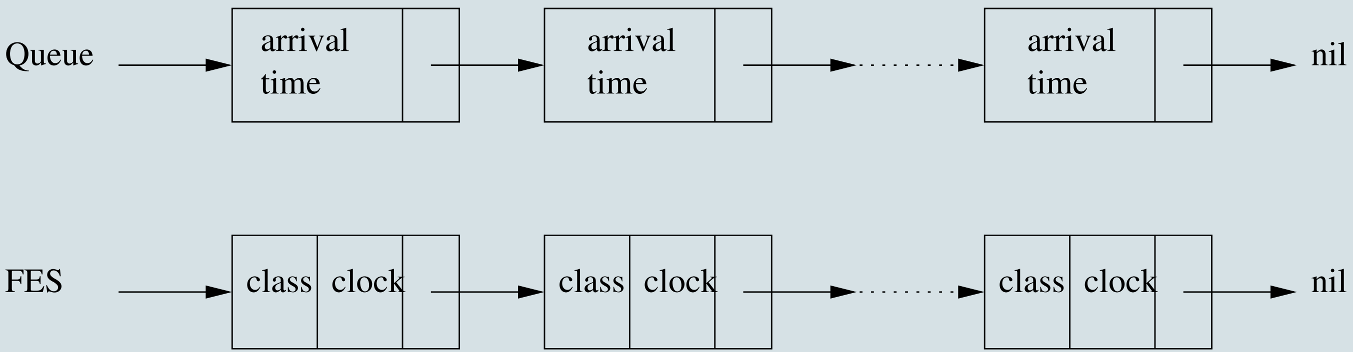

timing routine
select next event from FES
advance time to next event time
execute event
time < run length?
yes
no
end simulation

Record Job = (arrival time, ..., successor address)

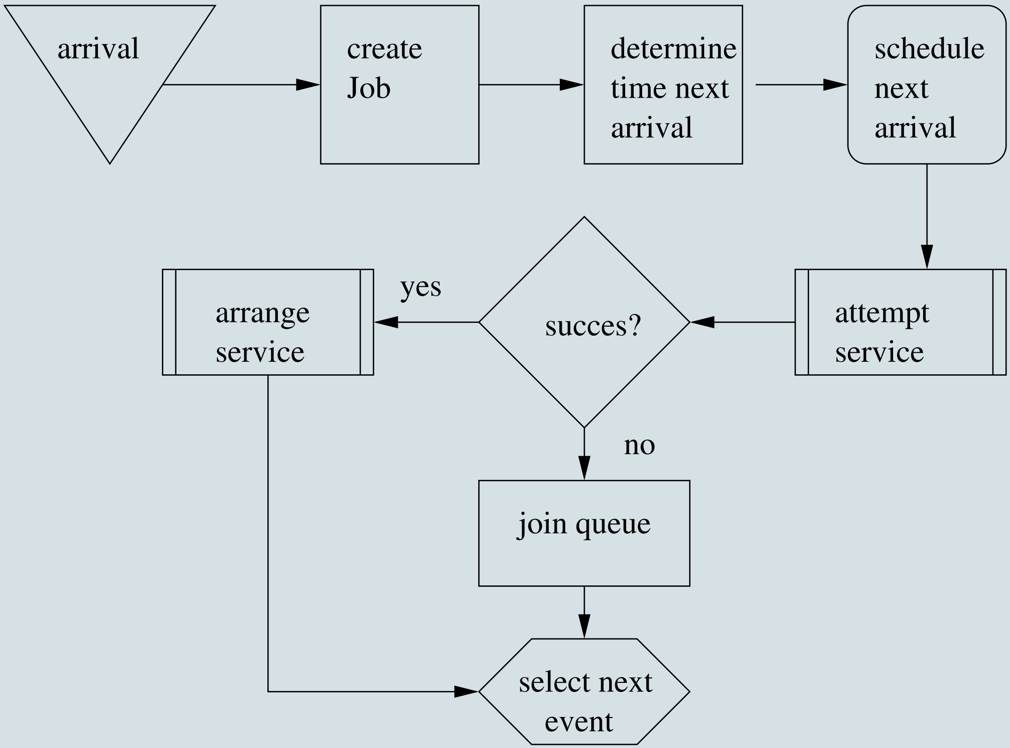
Record Event = (class, clock, ..., successor address)

The queue is a linked list of Job records ordered according to arrival time

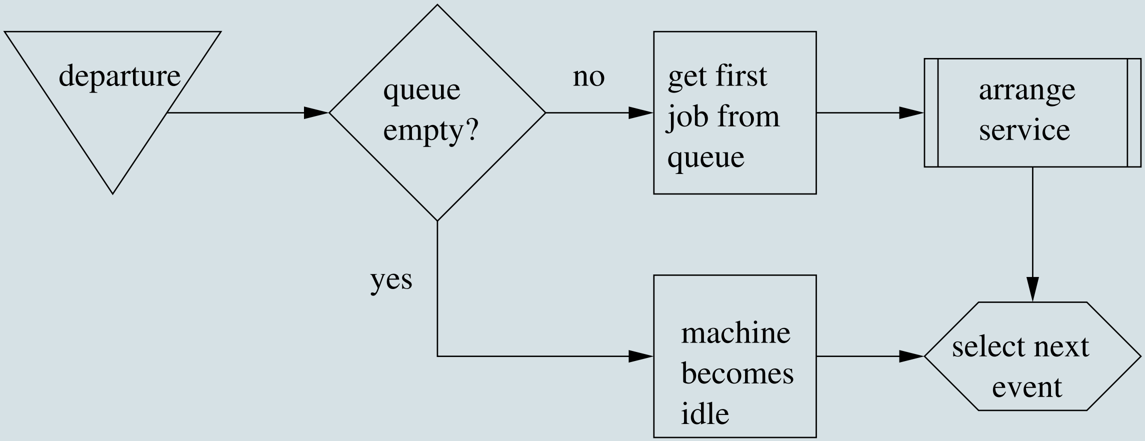
The FES is a linked list of Event records ordered according to clock time



Arrival event:



Departure event:



Initialization

```
t = 0       {current time}
queue = nil {queue is empty}
generate and schedule first arrival
N = 0       {number of jobs processed}
sum_w = 0   {sum of waiting times of processed jobs}
```

Main program

```
while (t < run_length)
do
  determine next_event
  t = event_time
  case next_event of
    arrival_event:
      generate and schedule next arrival
      if machine = busy
      then create and add job to queue
      else
        machine = busy
        N = N + 1
        generate and schedule next departure
```

```
    departure_event:
      if queue not empty
      then
        get first job from queue
        N = N + 1
        sum_w = sum_w + waiting_time
        generate and schedule next departure
      else machine = idle
end
```

Output

```
Mean waiting time = sum_w / N
```

Implementation in C

Definition of records: Events and Jobs

```
typedef struct job {
    double  arrival_time;
    struct job *next_job;
}

        job;

typedef struct event {
    int     class;
    double  clock;
    struct event *next_event;
}

        event;

event  *FES,          /* linked list of events */
       *Used_events;  /* linked list of used event notices */
job    *Queue,        /* linked list of jobs */
       *Used_jobs;    /* linked list of used job records */
```

Operations on the FES: create and destroy

```
event  *create_event()
{
    event  *temp;

    if (Used_events == NIL)
        return (event *) malloc(sizeof(event));
    else {
        temp = Used_events;
        Used_events = Used_events->next_event;
        return temp;
    }
}

void    destroy_event(event * pntr)
{
    pntr->next_event = Used_events;
    Used_events = pntr;
}
```

Operations on the FES: next and add

```
event  *next_event()
{
    event  *pntr;

    if (FES == NIL)
        return NIL;                              /* FES is empty */
    else {
        pntr = FES;
        FES = FES->next_event;
        return pntr;
    }
}
```

```
void    add_event(event * pntr)
{
    event  *link,
           *prev;

    if (FES == NIL) {
        FES = pntr;
        FES->next_event = NIL;
    } else {
        if (pntr->clock <= FES->clock) {
            pntr->next_event = FES;
            FES = pntr;
        } else {
            prev = FES;
            link = FES->next_event;
            while (link != NIL && pntr->clock > link->clock) {
                prev = link;
                link = link->next_event;
            }
            prev->next_event = pntr;
            pntr->next_event = link;
        }
    }
}
```

Initialization

```
void    initialization()
{
    srand48(seed);

    t = 0.0;
    busy = FALSE;
    Queue = NIL;
    Used_jobs = NIL;

    /* initialize FES */
    FES = create_event();
    FES->class = ARRIVAL;
    FES->clock = interarrivaltime();
    FES->next_event = NIL;

    Used_events = NIL;

    N = 0;
    sum_w = 0.0;
}
```

Main program

```
main()
{
    event  *pntr;

    getinput();
    initialization();

    while (t < run_length) {
        pntr = next_event();
        t = pntr->clock;                        /* advance time */
        switch (pntr->class) {
        case ARRIVAL:
            arrival_event();
            break;
        case DEPARTURE:
            departure_event();
            break;
        case NIL:
            printf("FES is empty\n");
            exit(1);
            break;
        }
        destroy_event(pntr);
    }

    output();
}
```

Compound event Arrival

```
void    arrival_event()
{
    event  *pntr_event;
    job    *pntr_job;

    pntr_event = create_event();                /* schedule next arrival */
    pntr_event->class = ARRIVAL;
    pntr_event->clock = t + interarrivaltime();
    add_event(pntr_event);

    if (busy) {
        pntr_job = create_job();
        pntr_job->arrival_time = t;
        add_job(pntr_job);
        if (Queue == NIL)
            printf("queue is nil\n");
    } else {
        busy = TRUE;
        N ++;
        pntr_event = create_event();
        pntr_event->class = DEPARTURE;
        pntr_event->clock = t + servicetime();
        add_event(pntr_event);
    }
}
```

Compound event Departure

```
void    departure_event()
{
    double  waiting_time;
    event   *pntr_event;
    job     *pntr_job;

    if (Queue != NIL) {
        pntr_job = next_job();
        N ++;
        waiting_time = t - pntr_job->arrival_time;
        sum_w += waiting_time;
        destroy_job(pntr_job);
        pntr_event = create_event();            /* schedule next departure */
        pntr_event->class = DEPARTURE;
        pntr_event->clock = t + servicetime();
        add_event(pntr_event);
    } else                                      /* Queue is empty */
        busy = FALSE;
}
```