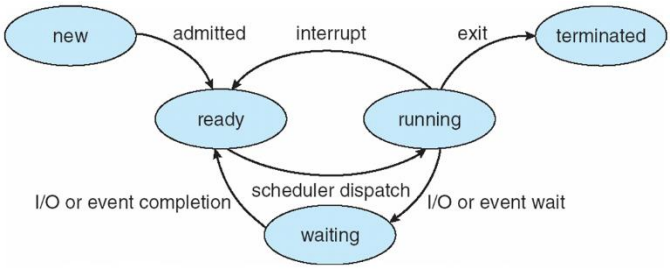
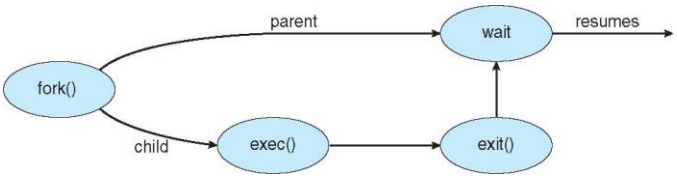
- **Goals of OS**
  - Excute user programs and make solving user problems easier.
  - Make the computer system convenient to use.
  - Use the computer hardware in an efficient manner.
- **Computer System**
  Hardware / OS / Application programs / Users
- **OS is**
  - Resource allocator
    Manages all resources. Decides between conflicting requests for efficient and fair resource use.
  - Control program
    Controls execution of programs to prevent errors and improper use of the computer
    언제 어떤 프로그램들이 CPU를 사용할 것인가.
- **Interrupt**
  Unexpected event from external device.
- **Exception**
  Unexpected event from CPU
- **Trap**
  Software interrupt. Usually using for system call.
- **Interrupt Vector**
  Contains the addresses of all the service routines.
- **Polling**
  Refers to actively sampling the status of an external device by client program as a synchronous activity.
  하나의 장치 또는 프로그램이 충돌 회피 또는 동기화 처리 등을 목적으로 다른 장치의 상태를 주기적으로 검사하여 일정한 조건을 만족하 때 송수신 등의 자료 처리를 하는 방식을 말한다.
- **Storage Definitions**
  - bit
    The basic unit of computer storage. Contains one of two values, 0 and 1.
  - byte = 8 bit
    Most computers use it smallest convenient chunk of storage.
  - word
    Computer architecture's native unit of data.
- **Direct Memory Access (DMA)**
  I/O 장치들이 메모리에 직접 접근하여 읽거나 쓸 수 있도록 하는 기능으로서, 컴퓨터 내부의 버스가 지원하는 기능이다. 대개의 경우에 메모리의 일정 부분이 DMA에 사용될 영역으로 지정되며, DMA가 지원되면 중앙처리장치가 데이터 전송에 관여하지 않아도 되므로 컴퓨터 성능이 좋아진다.
- **Clustered System**
  Like multiprocessor systems, but multiple systems working together.
  Usually sharing storage via a storage-area network (SAN). Some clusters are for high-performance computing (HPC). Some have distributed lock manage (DLM) to avoid conflicting operations.
- **Process**
  A program in execution. Process execution must progress in sequential fashion.
- **Diagram of Process State**



- **Process Creation**



- **Multithreading Models**
  Many-to-one / One-to-One / Many-to-Many / Two-level
- **Amdahl's Law**

$$\text{speedup} \leq \frac{1}{S + \frac{1-S}{N}}$$

  S is serial portion / N processing cores
  If application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times.
- **Dispatcher module**
  Gives control of the CPU to the process selected by the short-term scheduler. This involves: switching context / switching mode form kernel to user / jumping to the proper location in the user program to restart that program.
- **Dispatch latency** (≈ Switching context overhead)
  Time it takes for the dispatcher to stop one process and start another running.

- **CPU Utilization**
  Keep the CPU as busy as possible.
- **Throughput**
  Number of processes that complete their execution per time.
- **Turnaround Time**
  Amount of time to execute a particular process.
- **Waiting Time**
  Amount of time a process has been waiting in the ready queue.
- **Response Time**
  Amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment).
- **Scheduling Algorithm Optimization Criteria**
  Max: CPU utilization / throughput
  Min: turnaround time / waiting time / response time.
- **First-Come, First-Served (FCFS) Scheduling**

Proc.	P1	P2	P3
Burst T.	24	3	3

Suppose that the procs arrive in order: P1, P2, P3

P1	P2	P3
0	24	27

Waiting time for P1 = 0; P2 = 24; P3 = 27
Average waiting time = (0 + 24 + 27)/3 = 17

- **Convoy Effect**
  Short process behind long process is more efficient.

Suppose that the procs arrive in order: P2, P3, P1

P2	P3	P1
0	3	6

Waiting time for P1 = 6; P2 = 0; P3 = 3
Average waiting time = (6 + 0 + 3) / 3 = 3

- **Shortest-Job-First (SJF) Scheduling**

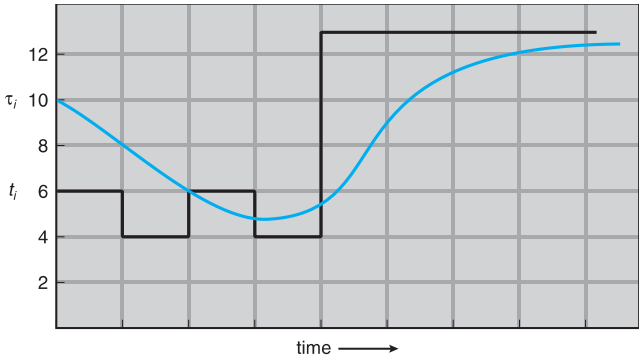
Proc.	P1	P2	P3	P4
Burst T.	6	8	7	3

P4	P1	P3	P2
0	3	9	16

Waiting time for P1 = 3; P2 = 16; P3 = 9; P4 = 0
Average waiting time = (3 + 16 + 9 + 0)/4 = 7

실제로 사용되지 않으며 얼마나 optical 한가 비교할 때 사용.

- **Determining Length of Next CPU Burst**
  t_n = actual length of n_{th} CPU burst
  τ_{n+1} = predicted value for the next CPU burst
  $\alpha, 0 \le \alpha \le 1$, commonly 0.5
  $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$
- **Shortest-remaining-time-first** (=Preemptive SJF)



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

If $\alpha = 0$, $\tau_{n+1} = \tau$. Recent history does not count.
If $\alpha = 1$, $\tau_{n+1} = \alpha\tau_n$. Only the actual last CPU burst cnt.

Proc.	P1	P2	P3	P4
Arrival T.	0	1	2	3
Burst T.	8	4	9	5

P1	P2	P4	P1	P3
0	1	5	10	17

Waiting time for
P1 = 10 - 1 = 9
P2 = 1 - 1 = 0
P3 = 17 - 2 = 15
P4 = 5 - 3 = 2
Average waiting time = (9 + 0 + 15 + 2) / 4 = 6.5

- **Priority Scheduling**

Proc.	P1	P2	P3	P4	P5
Burst T.	10	1	2	1	5
Priority	3	1	4	5	2

P2	P5	P1	P3	P4
0	1	6	16	18

Waiting time for
P1 = 6; P2 = 0; P3 = 16; P4 = 18; P5 = 1
Avg. waiting time = (6 + 0 + 16 + 18 + 1)/5 = 8.2

- **Round Robin**

  Each process gets a small unit of CPU time (time quantum ***q***), usually 10-100ms (context switch time < 10us). 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.

  Timer interrupts every quantum to schedule next process.

  If ***q*** is large: FIFO, ***q*** is small: ***q*** must be larger than context switch time.

Proc.	P1	P2	P3
Burst T.	24	3	3

P1	P2	P3	P1	P1	P1	P1	P1	P1	P1
0	3	6	9	12	15	18	21	24	27

Typically, higher average turnaround than SJF, but better response.

- **Differences between Process and Thread**

  프로세스는 독립된 메모리 공간을 가지고 코드와 데이터를 해당 메모리 공간에 올려 실행되고 있는 프로그램을 의미한다.

  스레드는 프로세스 안에 존재하며 독립된 스택 및 CPU 레지스터 정보를 갖는다. 프로세스 내의 다른 스레드들과 전역 data 및 code를 공유한다.

  프로세스는 자신의 코드와 데이터를 위한 독립된 주소 공간을 커널에서 할당 받아야 하므로 생성시 많은 자원을 필요로 한다. 이제 반하여 스레드는 프로세스 공간 내부에서 스택 및 CPU 레지스터 정보 등 만을 저장할 공간만 할당 받으면 되므로 생성시 상대적으로 적은 자원을 소모한다.

  프로세스간의 통신을 위해서는 커널의 도움을 통한 메시지 교환이나 공유 메모리 영역을 지정해주어야 하나, 스레드는 기본적으로 전역 데이터 영역을 공유하므로 스레드 간의 통신이 프로세스간의 통신 보다 좀 더 적은 자원을 필요로 한다.

- **Four Conditions for Deadlock**

  상호배제 (Mutual Exclusion): 프로세스들이 필요로 하는 자원에 대해 배타적인 통제권을 요구한다. 한번에 하나의 프로세스 만이 자원을 사용할 수 있다.

  점유대기 (Hold and Wait): 프로세스가 할당된 자원을 가진 상태에서 다른 자원을 기다린다.

  비선점 (No Preemption): 프로세스가 어떤 자원의 사용을 끝낼 때까지 그 자원을 뺏을 수 없다.

  순환대기 (Circular Wait): 각 프로세스는 순환적으로 다음 프로세스가 요구하는 자원을 가지고 있다.

- **test_and_set**

```
boolean test_and_set (Boolean *target){
        boolean rv = *target;
        *target = TRUE;
        return rv;
}
```

- **compare_and_swap**

```
int compare_and_swap(int *value, int expected, int new_value){
        int temp = *value;
        if ( *value == expected )
                *value = new_value;
        return temp;
}
```

- **Bounded-waiting Mutual Exclusion wit test_and_set**

```
do {  waiting[i] = TRUE;
        key = TRUE;
        while( waiting[i] && key )
                key = test_and_set(&lock);
        waiting[i] = FALSE;
        // cs
        while( (j != i) && !waiting[j] ) j = (j+1) % n;
        if( j == I ) lock = FALSE;
        else waiting[j] = FALSE;
} while(TRUE);
```

- **Semaphore with no Busy waiting**

```
typedef struct {
        int value;
        struct process *list;
} semaphore;

wait(swmaphore *S){
        S->value--;
        if ( S->value < 0 ) {
                add this process to S->list;
                block();
        }
}
signal(semaphore *S){
        S->value++;
        if ( S->value <= 0){
                remove a process P from S->list;
                wakeup(P);
        }
}
```

- **Bounded Buffer Problem Producer**

```
do {
        // produce an item
        wait(empty);
        wait(mutex);
        // add next produced to the buffer
        signal(mutex);
        signal(full);
} while(TRUE)
```

- **Bounded Buffer Problem Consumer**

```
do {  wait(full);
        wait(mutex);
        // remove an item from buffer
        signal(mutex);
        signal(empty);
        // consume the item
} while(TRUE);
```

- **Readers-Writers Problem Writer**

```
do {  wait(rw_mutex);
        // write
        signal(rw_mutex);
} while(TRUE);
```

- **Readers-Writers Problem Reader**

```
do {  wait(mutex);
        read_count++;
        if ( read_count == 1 )
                wait(rw_mutex);
        signal(mutex);
        // read
        wait(mutex);
        read_count--;
        if ( read_count == 0 )
                signal(rw_mutex);
        signal(mutex);
} while(TRUE);
```

- **Solution to Dining Philosophers**

```
monitor DiningPhilosophers {
        enum { THINKING, HUNGRY, EATING } state[5];
         condition self[5];
         void pickup(int i){
                  state[i] = HUNGRY;
                  test(i);
                  if ( state[i] != EATING )
                           self[i].wait;
        }
         void putdown(int i){
                  state[i] = THINKING;
                  test((i+4) % 5);
                  test((i+1) % 5);
        }
         void test(int i){
                  if( (state[(i+4)%5] != EATING
                  && (state[i] == HUNGRY)
                  && (state[(i+1)%5] != EATING ){
                           state[i] = EATING;
                           self[i].signal();
                  }
        }
         initialization_code(){
                  for( int i = 0; i < 5; i++)
                           state[i] = THINKING;
        }
}
DiningPhilosophers.pickup(i);
// EAT
DiningPhilosophers.putdown(i);
```

- a