Process Concept

- **Process**: a program in execution
  - process execution must progress in sequential fashion.

- A program is a passive entity, whereas a process is an active entity with a program counter and a set of associated resources.

- Each process has its own address space:
  - Text section (text segment) contains the executable code
  - Data section (data segment) contains the global variables
  - Stack contains temporary data (local variables, return addresses..)
  - A process may contain a heap, which contains memory that is dynamically allocated at run-time.

- The program counter and CPU registers are part of the process context.
Process

- Introduced to obtain a systematic way of monitoring and controlling program execution
- At first:
  - The unit that can be dispatched (scheduled)
  - The unit that ‘owns’ resources
    (This view changed later on with the advent of threads…)
- A process is an executable program with:
  - associated data (variables, buffers…)
  - execution context

Processes

- Multiprogramming of four programs
- Conceptual model of 4 independent, sequential processes
- Only one program active at any instant
OS Requirements for Processes

- OS must **interleave** the execution of several processes to maximize CPU usage while providing reasonable response time.
- OS must allocate resources to processes while **avoiding deadlock**.
- OS must support **inter process communication** and user creation of processes.

Process Creation

- Principal events that cause process creation:
  - System **initialization**
  - Execution of a process creation **system call** by a running process
  - **User** request to create a new process
Process Creation (Cont.)

- Parent process creates child processes, which, in turn create other processes, forming a tree (hierarchy) of processes.

- Issues
  - Will the parent and child execute concurrently?
  - How will the address space of the child be related to that of the parent?
  - Will the parent and child share some resources?

An example process tree
Process Creation in Unix

- Each process has a process identifier (pid).
- The parent executes fork() system call to spawn a child.
- The child process has a separate copy of the parent’s address space.
- Both the parent and the child continue execution at the instruction following the fork() system call. The return code for the fork() system call is
  - zero for the new (child) process
  - the (nonzero) pid of the child for the parent.
- Typically, the child executes a system call like execlp() to load a binary file into memory.

Example program with “fork”

```c
void main ()
{
    int pid;

    pid = fork();
    if  (pid < 0) {error_msg}
    else if (pid == 0) {/* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        exit(0);
    }
}
```
A very simple shell

```c
while (1)
{
    type_prompt();
    read_command(com);
    pid = fork();
    if (pid < 0) {error_msg}
    else if (pid == 0) { /* child process */
        execute_command(com);
    } else { /* parent process */
        wait(NULL);
    }
}
```

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int number = 7;
int main()
{
    pid_t pid;
    printf("Running the fork example\n");
    printf("The initial value of number is %d\n", number);
    pid = fork();
    printf(" PID is %d \n", pid);
    if (pid == 0){
        number = number;
        printf("In the child, the number is %d  PID is %d\n", number, pid);
        return 0;
    }
    else if (pid > 0) {
        wait(NULL);
        printf("In the parent, the number is %d\n", number);
    }
} // End forkeexample1.c
```

What happens to the value of number?
results

/forkexample1.exe

Running the fork example
The initial value of number is 7
PID is 2137
PID is 0

In the child, the number is 49  PID is 0
In the parent, the number is 7

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int number = 7;
int main()
{
    pid_t pid;
    printf("Running the fork example\n");
    printf("The initial value of number is \%d\n", number);
    pid = fork();
    printf(" PID is \%d \n", pid);

    if (pid == 0){
        number = number;
        fork();
        printf("In the child, the number is \%d  PID is \%d\n", number, pid);
        return 0;
    }
    else if (pid > 0) {
        wait(NULL);
        printf("In the parent, the number is \%d\n", number);
    }

}// End forxexample2.c
results

./forkexample2.exe

Running the fork example
The initial value of number is 7
PID is 2164
PID is 0
    In the child, the number is 49  PID is 0
    In the child, the number is 49  PID is 0
In the parent, the number is 7
Results

./execlexample.exe
Running execl code
PID is 2179
PID is 0

In the execl child, PID is 0

Running the fork example
The initial value of number is 7
PID is 2180
PID is 0

In the child, the number is 49, PID is 2181
In the child, the number is 49, PID is 0
In the parent, the number is 7
In the parent, done waiting

Process Termination

- Process executes last statement and asks the operating system to delete it (exit)
  - Output data from child to parent (via wait or waitpid).
  - Process’ resources are de-allocated by operating system

- Parent may terminate execution of children processes (e.g., TerminateProcess() in Win32)

- Process may also terminate due to errors

- Cascading termination – when a system does not allow a child process to continue after the parent has terminated.
Reasons for Process Termination (1)

- Normal completion
- Time limit exceeded
- Memory unavailable
- Memory bounds violation
- Protection error
  - example: write to read-only file
- Arithmetic error
- Time overrun
  - process waited longer than a specified maximum for an event

Reasons for Process Termination (2)

- I/O failure
- Invalid instruction
  - happens when try to execute data
- Privileged instruction
- Operating system intervention
  - such as when deadlock occurs
- Parent request to terminate one offspring
- Parent terminates so child processes terminate
Process States (Simplified)

- **Running** state
- **Ready** state
- **Blocked** state
- **New** state
  - OS has performed the necessary actions to *create* the process but has not yet admitted the process.
- **Exit** state
  - Termination moves the process to this state
  - Tables and other info are temporarily preserved for auxiliary program

A Five-state Process Model

Ready to exit: A parent may terminate a child process
Swapping/Suspending

- Processes may need to be \textit{swapped} out to disk.
  - This is true even with virtual memory!

- 2 new states:
  - \textit{Blocked Suspend}: blocked processes which have been swapped out to disk
  - \textit{Ready Suspend}: ready processes which have been swapped out to disk
Additional State Transitions

- **Blocked --> Blocked Suspend**
  - When all processes are blocked, the OS will make room to bring a ready process in memory

- **Blocked Suspend --> Ready Suspend**
  - When the event for which it has been waiting occurs

- **Ready Suspend --> Ready**
  - When no more ready process in main memory

- **Ready --> Ready Suspend**
  - When there are no blocked processes and must free memory for adequate performance

A Seven-state Process Model
Process Scheduling

- The operating system is responsible for managing the *scheduline* activities.
  - A uniprocessor system can have only one running process at a time
  - The main memory cannot always accommodate all processes at run-time
  - The operating system will need to decide on which process to execute next (*CPU scheduling*), and which processes will be brought to the main memory (*job scheduling*)

Process Scheduling Queues

- **Job queue** – set of all processes in the system.
- **Ready queue** – set of all processes residing in main memory, ready and waiting for CPU.
- **Device queues** – set of processes waiting for an I/O device.
- Process migration is possible among these queues.
Ready Queue and I/O Device Queues

Process Lifecycle
Schedulers

- The processes may be first spooled to a mass-storage system, where they are kept for later execution.

- The long-term scheduler (or job scheduler) – selects processes from this pool and loads them into memory for execution.
  - The long term scheduler, if it exists, will control the degree of multiprogramming.

- The short-term scheduler (or CPU scheduler) – selects from among the ready processes, and allocates the CPU to one of them.
  - Unlike the long-term scheduler, the short-term scheduler is invoked very frequently.

CPU and I/O Bursts

- CPU–I/O Burst Cycle –
  - Process execution consists of a cycle of CPU execution and I/O wait.

- 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-bound and I/O-bound Processes

(a) A CPU-bound process
(b) An I/O-bound process

Scheduler Impact

- Consequences of using I/O-bound and CPU-bound process information
  - Long-term (job) scheduler decisions
  - Short-term (CPU) scheduler decisions
Addition of Medium-Term Scheduler

- The medium-term scheduler can reduce the degree of multiprogramming by removing processes from memory.
- At some later time, the process can be re-introduced into memory (swapping).

Process Communication

- Mechanism for processes to communicate and to synchronize their actions.
- Two models
  - Communication through a shared memory region
  - Communication through message passing
Observe: in a distributed system, message-passing is the only possible communication model.

**Communication through message passing**

- Message system – processes communicate with each other *without resorting to shared variables*.

- A message-passing facility must provide at least two operations:
  - `send(message, recipient)`
  - `receive(message, recipient)`

- With *indirect communication*, the messages are sent to and received from *mailboxes* (or, *ports*).
  - `send (A, message)` /* A is a mailbox */
  - `receive (A, message)`
Communication through Message Passing

- Message passing can be either blocking (synchronous) or non-blocking (asynchronous)
  - Blocking Send: The sending process is blocked until the message is received by the receiving process or by the mailbox
  - Non-blocking Send: The sending process resumes the operation as soon as the message is received by the kernel
  - Blocking Receive: The receiver blocks until the message is available
  - Non-blocking Receive: “Receive” operation does not block; it either returns a valid message or a default value (null) to indicate a non-existing message

Communication through Shared Memory

- The memory region to be shared must be explicitly defined
- Using system calls – in Unix:
  - `Shmget` creates a shared memory block
  - `Shmat` maps an existing shared memory block into a process’s address space
  - `Shmdt` removes (“unmaps”) a shared memory block from the process’s address space
  - `Shmctl` is a general-purpose function allowing various operations on the shared block (receive information about the block, set the permissions, lock in memory, …)
- Problems with simultaneous access to the shared variables
- Compilers for concurrent programming languages can provide direct support when declaring variables (e.g. “shared int buffer”)
Shared Memory Example

```c
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{

    pid_t pid;
    /* the identifier for the shared memory segment */
    int segment_id;
    /* a pointer to the shared memory segment */
    char *shared_memory;
    /* the size (in bytes) of the shared memory segment */
    const int segment_size = 496;

    /* allocate a shared memory segment */
    segment_id = shmget(IPC_PRIVATE, segment_size, S_IRUSR | S_IWUSR);

    /* attach the shared memory segment */
    shared_memory = (char *) shmat(segment_id, NULL, 0);
    printf("Shared memory segment %d attached at address %p\n", segment_id, shared_memory);

    /* write a message to the shared memory segment */
    printf(shared_memory, "Hi there CS 571!\n");

    if (pid == 0) {
        printf("In the child, PID is %d memory is %p\n", pid, shared_memory);
        execlp("/sharechild.exe", "sharechild.exe", shared_memory, NULL);
        return 0;
    }

    else if (pid > 0) {
        wait(NULL);
        printf("In the parent, done waiting\n");
    }

    /* now print out the string from shared memory */
    printf("In Parent===>%s\n", shared_memory);

    /* now detach the shared memory segment */
    if (shmdt(shared_memory) == -1) {
        fprintf(stderr, "Unable to detach\n");
    }

    /* now remove the shared memory segment */
    shmmunmap(segment_id, IPC_RMID, NULL);

    return 0;
}
```

Shared Memory Example

```c
/* Do a fork */
pid = fork();

if (pid == 0) {
    printf("In the child, PID is %d memory is %p\n", pid, shared_memory);
    execlp("/sharechild.exe", "sharechild.exe", shared_memory, NULL);
    return 0;
}

else if (pid > 0) {
    wait(NULL);
    printf("In the parent, done waiting\n");
}

/* now print out the string from shared memory */
printf("In Parent===>%s\n", shared_memory);

/* now detach the shared memory segment */
if (shmdt(shared_memory) == -1) {
    fprintf(stderr, "Unable to detach\n");
}

/* now remove the shared memory segment */
shmmunmap(segment_id, IPC_RMID, NULL);
return 0;
```
sharechild.c Code

#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    //**Print out the string from shared memory */
    printf("\nFrom Child==>\$s\n\n", argv[1]);
    return 0;
}

Output

./share.exe
shared memory segment 720896 attached at address 0x10b74c000

    In the execl child,  PID is 0  memory is 0x10b74c000

    From Child==>Hi there  CS 571!
    *

    In the parent, done waiting

    In Parent==>Hi there  CS 571!
    *
Threads

- Overview
- Multithreading
- Example Applications
- User-level Threads
- Kernel-level Threads
- Hybrid Implementations
Threads

- A process, as defined so far, has only one thread of execution.

- **Idea:** Allow multiple threads of execution within the same process environment, to a large degree independent of each other.

- Multiple threads running in parallel in one process is analogous to having multiple processes running in parallel in one computer.

Threads (Cont.)

- Multiple threads within a process will share
  - The address space
  - Open files
  - Other resources

- Potential for efficient and close cooperation
Single and Multithreaded Processes

- When a multithreaded process is run on a single CPU system, the threads take turns to run.
- All threads in the process have exactly the same address space.

**Per Process Items**
- Address Space
- Global Variables
- Open Files
- Accounting Information

**Per Thread Items**
- Program Counter
- Registers
- Stack
- State

Multithreading
Multithreading (Cont.)

- Each thread can be in any one of the several states, just like processes.
- Each thread has its own stack.

Benefits

- Responsiveness
  - Multithreading an interactive application may allow a program to continue running even if part of it is blocked or performing a lengthy operation.
- Resource Sharing
  - Sharing the address space and other resources may result in high degree of cooperation
- Economy
  - Creating / managing processes is much more time consuming than managing threads.
- Better Utilization of Multiprocessor Architectures
Example Multithreaded Applications

- A multithreaded web server

![Diagram of a multithreaded web server]

- The outline of the code for the dispatcher thread (a), and the worker thread (b).

```c
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

```c
while (TRUE) {
    wait_for_work(&buf);
    check_cache(&buf, &page);
    if_not_in_cache(&page)
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(a)                                                  (b)
Threads in Multicore Platforms

- Concurrent and parallel execution of threads

Threads in Multicore Platforms (Cont.)

- Challenge: modify old programs and design new programs that are multithreaded

- Issues:
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging
Implementing Threads

- Processes usually start with a single thread
- Usually, library procedures are invoked to manage threads
  - `Thread_create`: typically specifies the name of the procedure for the new thread to run
  - `Thread_exit`
  - `Thread_join`: blocks the calling thread until another (specific) thread has exited
  - `Thread_yield`: voluntarily gives up the CPU to let another thread run
- Threads may be implemented in the user space or in the kernel space

User-level Threads

- User threads are supported above the kernel and are implemented by a thread library at the user level.
- The library (or run-time system) provides support for thread creation, scheduling and management with no support from the kernel.
User-level Threads (Cont.)

- When threads are managed in user space, each process needs its own private thread table to keep track of the threads in that process.
- The thread-table keeps track only of the per-thread items (program counter, stack pointer, register, state..)
- When a thread does something that may cause it to become blocked locally (e.g. wait for another thread), it calls a run-time system procedure.
- If the thread must be put into blocked state, the procedure performs thread switching.

User-level Threads: Advantages

- The operating system does not need to support multi-threading.
- Since the kernel is not involved, thread switching may be very fast.
- Each process may have its own customized thread scheduling algorithm.
- Thread scheduler may be implemented in the user space very efficiently.
User-level Threads: Problems

- The implementation of blocking system calls is highly problematic (e.g. read from the keyboard). All the threads in the process risk being blocked!

- Possible Solutions:
  - Change all system calls to non-blocking
  - Sometimes it may be possible to tell in advance if a call will block (e.g. `select` system call in some versions of Unix) → “jacket code” around system calls

- How to deal with page faults?

Kernel-level threads

- Kernel threads are supported directly by the OS: The kernel performs thread creation, scheduling and management in the kernel space
Kernel-level threads

- The kernel has a thread table that keeps track of all threads in the system.
- All calls that *might* block a thread are implemented as system calls (greater cost).
- When a thread blocks, the kernel may choose another thread from the same process, or a thread from a different process.

Hybrid Implementations

- An alternative solution is to use kernel-level threads, and then multiplex user-level threads onto some or all of the kernel threads.
- A kernel-level thread has some set of user-level threads that take turns using it.
Pthreads

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization.
- API specifies behavior of the thread library, implementation is up to development of the library.
- Common in UNIX operating systems
- Pthread programs use various statements to manage threads: `pthread_create`, `pthread_join`, `pthread_exit`, `pthread_attr_init`, ...

<table>
<thead>
<tr>
<th>Thread Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pthread_create</code></td>
<td>Create a new thread in the caller’s address space</td>
</tr>
<tr>
<td><code>pthread_exit</code></td>
<td>Terminate the calling thread</td>
</tr>
<tr>
<td><code>pthread_join</code></td>
<td>Wait for a thread to terminate</td>
</tr>
<tr>
<td><code>pthread_mutex_init</code></td>
<td>Create a new mutex</td>
</tr>
<tr>
<td><code>pthread_mutex_destroy</code></td>
<td>Destroy a mutex</td>
</tr>
<tr>
<td><code>pthread_mutex_lock</code></td>
<td>Lock a mutex</td>
</tr>
<tr>
<td><code>pthread_mutex_unlock</code></td>
<td>Unlock a mutex</td>
</tr>
<tr>
<td><code>pthread_cond_init</code></td>
<td>Create a condition variable</td>
</tr>
<tr>
<td><code>pthread_cond_destroy</code></td>
<td>Destroy a condition variable</td>
</tr>
<tr>
<td><code>pthread_cond_wait</code></td>
<td>Wait on a condition variable</td>
</tr>
<tr>
<td><code>pthread_cond_signal</code></td>
<td>Release one thread waiting on a condition variable</td>
</tr>
</tbody>
</table>
```c
#include <pthread.h>
#include <stdio.h>

int sum; /* shared */

void *runner(void *param) {
    int i, upper = atoi(param);
    sum = 0;
    for (i=1; i<upper; i++)
        sum += i;
    pthread_exit(0);
}

int main(int argc, char*argv[]) {
    pthread_t tid;
    pthread_attr_t attr;
    if (argc != 2) {
        fprintf(stderr, "usage: a.out <int>\n");
        return -1;
    } else if (atoi(argv[0]) < 0) {  
        fprintf(stderr, "%d must be >= 0\n", atoi(argv[0]));
        return -1;
    }
    pthread_attr_init(&attr);
    pthread_create(&tid, &attr, runner, argv[1]);
    pthread_join(tid, NULL);
    printf("sum = %d\n", sum);
    return 0;
}
```

### Windows XP Threads

- Windows XP supports kernel-level threads
- The primary data structures of a thread are:
  - **ETHREAD** (executive thread block)
    - Thread start address
    - Pointer to parent process
    - Pointer to the corresponding KTHREAD
  - **KTHREAD** (kernel thread block)
    - Scheduling and synchronization information
    - Kernel stack (used when the thread is running in kernel mode)
    - Pointer to TEB
  - **TEB** (thread environment block)
    - Thread identifier
    - User-mode stack
    - Thread-local storage
Linux Threads

- In addition to `fork()` system call, Linux provides the `clone()` system call, which may be used to create threads.

- Linux uses the term *task* (rather than process or thread) when referring to a flow of control.

- A set of flags, passed as arguments to the `clone()` system call determine how much sharing is involved (e.g. open files, memory space, etc.)