Processes

- Process Concept
- Process States
- Process Creation and Termination
- Process Scheduling
- Process Communication
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.
The Process (Cont.)

- 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 States

As a process executes, it changes state:

- **New**: The process is being created.
- **Running**: Instructions are being executed.
- **Waiting (blocked)**: The process is waiting for some event to occur (such as I/O completion or receipt of a signal).
- **Ready**: The process is waiting to be assigned to the CPU.
- **Terminated**: The process has finished execution.
Simple State Transition Model

- New
- Ready
  - Admit
  - Event occurs
- Waiting
  - Event wait
- Running
  - Scheduler Dispatch
  - Timeout
- Terminated
  - Exit
To implement the process model, the operating system maintains the *process table*, with one *process control block* per process.

```
<table>
<thead>
<tr>
<th>process state</th>
</tr>
</thead>
<tbody>
<tr>
<td>process number</td>
</tr>
<tr>
<td>program counter</td>
</tr>
<tr>
<td>registers</td>
</tr>
<tr>
<td>memory limits</td>
</tr>
</tbody>
</table>
| list of open files | ...
```

GMU CS 571
Process Control Block in Unix

- **Process Management**
  - Registers
  - Program Counter
  - Program Status Word
  - Stack Pointer
  - Process State
  - Priority
  - Scheduling Parameters
  - Process ID
  - Parent process
  - Process group
  - Time when process started
  - CPU time used

- **Memory Management**
  - Pointer to text (code) segment
  - Pointer to data segment
  - Pointer to stack segment

- **File Management**
  - Root directory
  - Working directory
  - User Id
  - Group Id
CPU Switch From Process to Process

- **Process $P_0$**
  - Executing
  - Interrupt or system call
  - Save state into PCB
  - Reload state from PCB

- **Operating System**

- **Process $P_1$**
  - Idle
  - Executing
  - Save state into PCB
  - Reload state from PCB

GMU CS 571
Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process.

- Context-switch time is pure overhead; the system does no useful work while switching.

- Overhead dependent on hardware support (typically 1-1000 microseconds).
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 & Challenges
  - 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 in Solaris

```
GMU CS 571

init
pid = 1

inetd
pid = 140

telnetdaemon
pid = 7776

Csh
pid = 7778

Netscape
pid = 7785

emacs
pid = 8105

pageout
pid = 2

dtlogin
pid = 251

Xsession
pid = 294

tsdt_shel
pid = 340

Csh
pid = 1400

ls
pid = 2123

cat
pid = 2536

fsflush
pid = 3

Sched
pid = 0
```
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);
    }
}
```
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);
    }
}
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 (virtual memory resources, locks, etc.) are deallocated by operating system.
  - The OS records the process status and resource usage, notifying the parent in response to a wait call.

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

- Process may also terminate due to errors
Process Termination (Cont.)

- In Unix, a process does not completely release its resources after termination until its parent waits for it.

- If a parent is not waiting when the process terminates (but keeps running), that process becomes a zombie.

- If a parent terminates without waiting for children, the child processes become orphans.

- When a process terminates, its orphaned children and zombies are adopted by a special system process init (with pid = 1), that periodically waits for children.
Process Scheduling

The operating system is responsible for managing the scheduling 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 between these queues.
Ready Queue and I/O Device Queues
Process Lifecycle

- ready queue
- CPU
- I/O
- I/O queue
- I/O request
- time slice expired
- child executes
- fork a child
- interrupt occurs
- wait for an interrupt
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
What is the impact of the degree of multiprogramming on the CPU utilization?

Suppose average process computes only 20% of time it is sitting in memory and does I/O 80% of time.

We tend to say: With five processes CPU should be busy, all the time.

Is it correct?

Probabilistic view: CPU Utilization = 1 – xn where x is the average ratio of I/O time to process lifetime in memory, and n is the degree of multiprogramming.
CPU utilization as a function of the number of processes in memory.

[Diagram showing CPU utilization in percent as a function of the degree of multiprogramming, with curves for 20%, 50%, and 80% I/O wait.]
Scheduler Impact

Consequences of using I/O-bound and CPU-bound process information:

- Long-term (job) scheduler decisions
- Short-term (CPU) scheduler decisions
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
Communication Models

(a) Message Passing

(b) Shared Memory
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)
  - These operations can be either blocking (synchronous) or non-blocking (asynchronous).
- Observe: in a distributed system, message-passing is the only possible communication model.
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.
Multiple threads within a process will share:
- The address space
- Open files
- Other resources

Potential for efficient and close cooperation
Single and Multithreaded Processes

- Single-threaded process
- Multithreaded process

Diagrams showing the components of code, data, files, registers, and stack for both single-threaded and multithreaded processes.
Multithreading

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

<table>
<thead>
<tr>
<th>Per Process Items</th>
<th>Per Thread Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address Space</td>
<td>Program Counter</td>
</tr>
<tr>
<td>Global Variables</td>
<td>Registers</td>
</tr>
<tr>
<td>Open Files</td>
<td>Stack</td>
</tr>
<tr>
<td>Accounting Information</td>
<td>State</td>
</tr>
</tbody>
</table>
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 word-processor with three threads
  - Re-formatting
  - Interacting with user
  - Disk back-up

- What would happen with a single-threaded program?
Example Multithreaded Applications

A multithreaded web server
Example Multithreaded Applications

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)
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.
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.
- Some kernels recycle their threads, new threads use the data-structures of already completed threads.
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,…
Thread Calls in POSIX

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