Processes

• Hundreds of things going on in the system: how to manage?

• How to make things simple?
  – Decompose computation into separate processes

• How to make things reliable?
  – Isolate processes from each other to protect from each others’ faults

• How to speed up?
  – Overlap I/O bursts of one process with CPU bursts of another

Kavita Bala, Computer Science, Cornell University
What is a process?

- A program being executed
  - Sequential, one instruction at a time.

- An operating system abstraction: a thread of execution running in a restricted virtual environment – a virtual CPU and virtual memory environment, interfacing with the OS via system calls.
  - The unit of execution
  - The unit of scheduling
  - Thread of execution + address space

The same as “job” or “task” or “sequential process”. Closely related to “thread”.

Process != Program

<table>
<thead>
<tr>
<th>Header</th>
<th>Code</th>
<th>Initialized data</th>
<th>BSS</th>
<th>Symbol table</th>
<th>Line numbers</th>
<th>Ext. refs</th>
<th>Executable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program is passive</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Code + static data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Process is running program |
| • stack, registers, heap, pc |

**Example:**
- We both run Firefox on one machine.
  - same program
  - separate processes
  - same virtual address space
  - different physical memory

Kavita Bala, Computer Science, Cornell University
Context Switch

- Context Switch
  - Process of switching CPU from one process to another
- State of a running process must be saved and restored:
  - Program Counter, Stack Pointer, General Purpose Registers
- Suspending a process: OS saves state
  - Saves register values
- To execute another process, the OS restores state
  - Loads register values

Details of Context Switching

- Context switching code is architecture-dependent
  - Depends on registers
- Very tricky to implement
  - OS must save state without changing state
  - Must run without changing any user program registers
    - CISC: single instruction saves all state
    - RISC: reserve registers for kernel
      - Or way to save a register and then continue
- Overheads: CPU is idle during a context switch
  - Explicit:
    - direct cost of loading/storing registers to/from main memory
  - Implicit:
    - Opportunity cost of flushing useful caches (cache, TLB, etc.)
    - Waiting for pipeline to drain in pipelined processors
How to create a process?

- Double click on a icon?
- After boot OS starts the first process
  - e.g. `init` for Linux, `ntoskrnl.exe` for XP
- The first process creates other processes:
  - the creator is called the parent process
  - the created is called the child process
  - the parent/child relationships creates a process tree

Processes Under UNIX

- New child process is created by the `fork()` system call:

```
int fork()
```
  - creates a new address space
  - copies the parent’s address space into the child’s
    - uses copy-on-write to avoid copying memory that is only read
  - starts a new thread of control in the child’s address space
  - parent and child are almost identical
    - in parent, `fork()` returns a non-zero integer
    - in child, `fork()` returns a zero.
    - difference allows parent and child to distinguish themselves
  - `fork()` returns TWICE!
Example

```c
int main(int argc, char **argv)
{
    char *myName = argv[1];
    int cpid = fork();
    if (cpid == 0) {
        printf("The child of %s is %d\n", myName, getpid());
        exit(0);
    } else {
        printf("My child is %d\n", cpid);
        exit(0);
    }
}
```

What does this program print?
Kavita Bala, Computer Science, Cornell University

Bizarre But Real

```
$ gcc a.c
$ ./a.out foobar
The child of foobar is 23874
My child is 23874
```

Kavita Bala, Computer Science, Cornell University
Parallel Programming and Synchronization

Cooperating Processes

- Processes can be independent or can work cooperatively
  - Cooperating processes exploit parallelism=concurrency
- Cooperating processes can be used for:
  - speedup by spreading computation over multiple processors/cores
  - speedup and improving interactivity: one process can work while others are stopped waiting for I/O.
  - better structuring of an application into separate concerns
    - e.g., a pipeline of processes processing data
- But: cooperating processes need ways to
  - Communicate information
  - Coordinate (synchronize) activities
Shared memory

- By default processes have disjoint physical memory -- complete isolation prevents communication

- Processes can set up a segment of memory as shared with other process(es)
  - Typically part of the memory of the process creating the shared memory. Other processes attach this to their memory space.

- Allows high-bandwidth communication between processes by just writing into memory

Example

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

int main(int argc, char **argv) {
    char* shared_memory;
    const int size = 4096;
    int segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
    int cpid = fork();
    if (cpid == 0) {
        shared_memory = (char*) shmat(segment_id, NULL, 0);
        sprintf(shared_memory, "Hi from process %d", getpid());
    } else {
        wait(NULL);
        shared_memory = (char*) shmat(segment_id, NULL, 0);
        printf("Process %d read: %s\n", getpid(), shared_memory);
        shmdt(shared_memory);
        shmctl(segment_id, IPC_RMID, NULL);
    }
}
```

Allocate shared memory, return handle

Attach shared memory to address space

Wait for forked process to finish

Detach shared memory from address space and deallocate
Processes are heavyweight

- Parallel programming with processes:
  - They share almost everything
  - They all share the same code and any data in shared memory (process isolation is not useful)
  - They all share the same privileges

- What don’t they share?
  - Each has its own PC, registers, and stack

- Idea: why don’t we separate the idea of process (address space, accounting, etc.) from that of the minimal “thread of control” (PC, SP, registers)?

Threads vs. processes

- Most operating systems therefore support two entities:
  - the process,
    - which defines the address space and general process attributes
  - the thread,
    - which defines a sequential execution stream within a process

- A thread is bound to a single process.
  - For each process, however, there may be many threads.
- Threads are the unit of scheduling
- Processes are containers in which threads execute
Multithreaded Processes

Two threads, one counter

Web servers use concurrency:
- Multiple threads handle client requests in parallel.
- Some shared state, e.g. hit counts:
  - each thread increments a shared counter to track number of hits

    ...  
    \[ \text{hits} = \text{hits} + 1; \]  
    ...  

- What happens when two threads execute concurrently?
Shared counters

- Usual result: works fine.
- Possible result: lost update!

\[
\begin{align*}
\text{hits} &= 0 \\
\text{time} \quad \rightarrow & \quad T1 \quad \rightarrow & \quad T2 \\
\text{read hits} &= (0) \\
\text{hits} &= 0 + 1 \\
\text{hits} &= 1 \\
\end{align*}
\]

- Occasional timing-dependent failure ⇒ Difficult to debug
- Called a race condition

Race conditions

- Def: a timing-dependent error involving shared state
  - Whether it happens depends on how threads scheduled: who wins “races” to instructions that update state
  - Races are intermittent, may occur rarely
    - Timing dependent = small changes can hide bug
  - A program is correct only if all possible schedules are safe
    - Number of possible schedule permutations is huge
    - Need to imagine an adversary who switches contexts at the worst possible time
Critical Sections

- Basic way to eliminate races: use *critical sections* that only one thread can be in
  - Contending threads must wait to enter

```
T1
CSEnter();
Critical section
CSExit();
T2
```

Mutexes

- Critical sections typically associated with mutual exclusion locks (*mutexes*)
- Only one thread can hold a given mutex at a time
- Acquire/Lock mutex on entry to critical section
  - Or block if another thread already holds it
- Release/Unlock mutex on exit
  - One waiting thread (if any) can acquire & proceed

```
pthread_mutex_init(m);
pthread_mutex_lock(m);
hits = hits+1;
pthread_mutex_unlock(m);
pthread_mutex_init(m);
pthread_mutex_lock(m);
hits = hits+1;
pthread_mutex_unlock(m);
```

Kavita Bala, Computer Science, Cornell University
Using atomic hardware primitives

- Mutex implementations usually rely on special hardware instructions that atomically do a read and a write.
- Requires special memory system support on multiprocessors

Mutex init: lock = false;

while (test_and_set(&lock));

Critical Section

lock = false;

test_and_set uses a special hardware instruction that sets the lock and returns the OLD value (true: locked; false: unlocked)
- Alternative instruction: compare & swap