Multiprogramming

- Assume we have two programs P and Q.
- Each is started up to make two processes p and q.
- It is not the case that process p has all its instructions finished before process q.
- Process p has an instruction that requires a read/write from/to disk (or terminal).
- Reading from disk is slow.
- Why not have instructions from q execute while p is waiting?
Multiprogramming

- Multiprogramming allows for the execution of multiple processes
- But only one process active at any time
Why Multiprogramming?

- Operating systems allow for **interleaved execution**
  - On a single-processor system, no more than one process ever runs.
  - However, one process’s instructions may be executed before the completion of the instructions from another process

- The **objective** is to have some process running at all times in order to maximize CPU utilization.
Process Switching

- Current process executes an I/O operation
- OS needs to be able to suspend current process so that another process can execute
- This is referred to as context switching
Process Switching

- OS needs to be able to suspend current process
- OS captures information about a process
- Information captured must be sufficient to restore the hardware to the same configuration it was in when the process was switched out.
Characterizing a Process

- Each process is represented in the OS by a process control block (PCB) which contains all the state for a program in execution including (but not limited to):
  - Pointer to text, data and stack segment information
  - An execution stack encapsulating the state of procedure calls
  - The program counter (PC) indicating the next instruction
  - Current values of the set of general-purpose registers
  - A set of operating system resources e.g., open files, network connections
  - Process identifier (PID)
  - Process priority (for scheduling purposes)
  - etc.
**Process Control Block (PCB)**

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

- Program instructions operate on operands in memory and (temporarily) in registers

Diagram:
- Memory
  - Prog1 Code
  - Prog1 Data
  - Prog2 Code
  - Prog2 Data
  - Prog2 State

- CPU
  - ALU
  - SP
  - PC

- Prog1 has CPU
- Prog2 is suspended

Load A1, R1
Load A2, R2
Add R1, R2, R3
Store R3, A3

...
Context Switching

- Saving all the information about a process allows a process to be temporarily suspended and later resumed from the same point.

Diagram:
- Memory:
  - Prog1 Code
  - Prog1 Data
  - Prog1 State
  - Prog2 Code
  - Prog2 Data
  - Prog2 State
- CPU:
  - ALU
  - SP PC
- OS suspends Prog1
Saving all the information about a process allows a process to be *temporarily suspended* and later *resumed*.
Context Switching

- Program instructions operate on operands in memory and in registers

Diagram:
- Memory
  - Prog1 Code
  - Prog1 Data
  - Prog1 State
- Prog2 Code
- Prog2 Data
- CPU
  - ALU
  - SP
  - PC
- Load A1, R1
- Load A2, R2
- Sub R1, R2, R3
- Store R3, A3
-...

Prog2 has CPU
Prog1 is suspended
The process control block is represented by the C structure `task_struct`

Some of the fields in `task_struct` include:

- `pid_t pid;` /* process identifier*/
- `long state;` /* state of the process*/
- `unsigned int time_slice;` /* scheduling information */
- `struct list_head children;` /* this process’s children */
- `struct files_struct *files;` /* list of open files */
Process Representation in Linux

- All active processes are represented using a doubly linked list of `task_struct`
- The kernel maintains a pointer, `current`, to the process currently executing on the system
- Example of the kernel manipulating one the fields in the `task_struct` is this:
  ```c
  current->state = new_state
  ```
Process Execution States

- As a process executes, it changes execution state.
- The execution state of a process is defined in part by the current activity of the process.
- A process may be in one of the following execution states:
  - **New**: The process is being created.
  - **Running**: Instructions are being executed.
  - **Waiting**: The process is waiting for some event to occur (such as an I/O completion or reception of signal).
  - **Ready**: The process is waiting to be assigned to a processor.
  - **Exit**: The process has finished executing.
- Only one process can be running on any processor at any instant.
- Many processes may be ready and waiting.
Process Execution States

When you run a program, a new process is created.

If the system has sufficient memory, then the new process is loaded into memory and placed in the ready queue.

CPU scheduler takes a process from the head of a ready queue to execute. (Sometimes, there may be multiple ready queues.)

When processes are scheduled in a round robin manner, then when time quantum expires, the process is returned to the ready queue.

If the wait for I/O is over, there is a return to the Ready state.

When the waiting is over:
- An interrupt is generated
- P ready queue

Process may be blocked by:
- I/O (wait for I/O to complete)
- Semaphore wait
- Sleep
- etc.

Event Wait

Event Occurs

Timeout

Dispatch

Admit

Exit

Running

New

Ready

Waiting
**Process Execution States**

**New**
- When you run a program, a new process is created.
- If the system has sufficient memory, then the new process is loaded into memory and placed in the ready queue.

**Ready**
- Admit

**Running**
- Timeout

**Exit**
- Dispatch

**Waiting**
- Event Occurs
- Event Wait
- CPU scheduler takes a process from the head of a ready queue to execute.
- (Sometimes, there may be multiple ready queues.)

**Process may be blocked by:**
- I/O (wait for I/O to complete)
- Semaphore wait
- Sleep
- etc.

When the wait is over:
- An interrupt is generated.
- When the wait for I/O is over, there is a return to the Ready state.
- When the waiting is over, there is a return to the Ready state.
Question

Why is there no arrow from waiting to running?

When should a process be preempted?
Scheduling

- The purpose of multiprogramming is to have a process running at all times.
- The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each process.
- The **process scheduler** selects an available process.
- There may be multiple processes to select from.
Scheduling Queues

- As processes enter the system, they are put into a **job queue**, which consists of all processes in the system.
- The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**.
- Queues are implemented using linked list.
- A ready queue header contains pointers to the first and last final process control blocks in the list.
- Each PCB includes a pointer field that points to the next PCB in the ready queue.
Other Queues

- When a process is allocated the CPU, it executes for a while and eventually quits, or interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request.

- The list of processes waiting for a particular I/O device is called a device queue.
Summary

- Discussed the need for multiprogramming
- Process representation