Scheduling processes

- OS keeps PCBs on different queues
  - Ready processes are on ready queue - OS chooses one to dispatch
  - Processes waiting for I/O are on appropriate device queue
- OS regulates PCB migration during life cycle of corresponding process

Why scheduling is interesting

- Which process should be dispatched?
- Processes are not created equal!
  - CPU-bound process: long CPU bursts
  - I/O-bound process: short CPU bursts

Context Switch

- Stopping a process and starting another
  - 100–10,000 per second, so must be fast

Metrics

- Batch systems
  - Maximize throughput (jobs completed/time observed)
  - Minimize turnaround time (time between submission and termination)
  - Maximize CPU utilization (time CPU busy/time observed)
- Interactive systems
  - Minimize response time (time from start to finish)
  - Proportionality - meet users’ expectations
**FCFS**
First Come First Served

- Processes $P_1, P_2, P_3$ with compute time 12, 3, and 3
- Job arrival $P_1, P_2, P_3$

**SJF**
Shortest Job First

- Schedule jobs in order of estimated completion† time
- Optimal* average turnaround time (att)

**Intuition:** $att = (r_1 + r_2 + r_3 + r_4 + r_5 + r_6) / 6$

*with preemption, remaining time  †when jobs are available simultaneously
**Shortest Job First (SJF)**

- Schedule jobs in order of estimated completion time
- Optimal* average turnaround time (att)
- Intuition: \( att = \frac{r_1 + r_2 + r_3 + r_4 + r_5 + r_6}{6} \)

**Priority Scheduling**

- Assign a number to each job and schedule jobs in (increasing) order
- Reduces to SJF if \( \tau_n \) is used as priority
- To avoid starvation, change job’s priority with time (aging)

\[ att = \frac{(r_1 + r_2) + (r_4 - c_3) + (r_5 - c_3) + r_3 + c_1 + c_5 + r_6}{6} \]
\[ = \frac{(r_1 + r_2 + r_3 + r_4 + r_5 + r_6 + (c_4 + c_5 - 2c_3))}{6} \]

*with preemption, remaining time  
*when jobs are available simultaneously
Round-Robin

- Each process is allowed to run for a quantum
- Context is switched (at the latest) at the end of the quantum
- Too long a quantum can reduce response time (to FCFS in the limit!)
- Too short a quantum can cause many unnecessary preemptions

MLF

Multi-level feedback queues

- Multiple priority levels, round robin within level
- Quantum size increases exponentially as priority decreases
- Jobs demoted to lower priority if they don’t complete within quantum
- Hard to get right in practice
  - how do we handle a process whose behavior changes over time?

Real-time scheduling

- A set of \( m \) periodic events is schedulable if
  \[ \sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1 \]
  where
  - \( C_i \) is the CPU time required to process event \( i \)
  - \( P_i \) is the period (1/rate) at which event \( i \) occurs
- Two dominant real-time scheduling policies:
  - **Rate-monotonic**
    - priority = period
  - **Deadline**
    - priority = release time + deadline

Rate monotonic

- Consider scheduling audio and video playout processes
  - audio processes 1 audio sample every 20 ms
  - video processes one video frame every 33 ms

Rate monotonic is guaranteed to work if
\[ \sum_{i=1}^{m} \frac{C_i}{P_i} \leq m(2^{1/m} - 1) \]
or, when \( m \to \infty \),
\[ \sum_{i=1}^{m} \frac{C_i}{P_i} \leq \ln 2 \]
Process Manipulation

- Basic process manipulation: creation, program loading, exiting, ...
- Example: Unix Operating system
  - Creation and deletion: `fork()`, `exec()`, `wait()`, `exit()`
  - Process signaling: `kill()`
  - Process control: `ptrace()`, `nice()`, `sleep()`

Deadline scheduling

- Consider scheduling audio and video playout processes
  - audio processes 1 audio sample every 20 ms
  - video processes one video frame every 33 ms
- In our case, deadline = inter-arrival time

Process Manipulation in Unix

- The system creates the first process (sysproc)
- The first process creates other processes in a tree like structure:
  - the creator is called the **parent** process
  - the created is called the **child** process
- Unix system interface includes a call to create processes
  - `fork()`
Unix’s fork()

- Creates a child process such that it inherits:
  - identical copy of all parent’s variables & memory
  - identical copy of all parent’s CPU registers (except one)
- Both parent and child execute at the same point after fork() returns:
  - for the child, fork() returns 0
  - for the parent, fork() returns the process identifier of the child
- Simple implementation of fork():
  - allocate memory for the child process
  - copy parent’s memory and CPU registers to child’s
  - Expensive !!

Can one reduce overhead without changing semantics?

Using Unix’s fork()

- The execution context for the child process is a copy of the parent’s context at the time of the call

Program Loading: exec()

- The exec() call allows a process to “load” a different program and start execution at _start
- It allows a process to specify the number of arguments (argc) and the string argument array (argv)
- If the call is successful
  - it is the same process ...
  - but it runs a different program !!
- Two implementation options:
  - overwrite current memory segments with the new values
  - allocate new memory segments, load them with the new values, and deallocate old segments

General Purpose Process Creation

In the parent process:
main()
...
int pid = fork();    // create a child
if(pid == 0) {       // child continues here
  exec("program", argc, argv0, argv1, ...);
}
else {                // parent continues here
  ...
}
Tying all together:
the Unix shell

while(! EOF)
read input
handle regular expressions
int pid = fork()   // create child
if (pid == 0) { // child here
  exec("program", argc, argv0,...);
} else { // parent here
  ... 
}

Orderly Termination:
exit()

- After the program finishes execution, it calls `exit()`
- This system call:
  - takes the “result” of the program as an argument
  - closes all open files, connections, etc.
  - deallocates memory
  - deallocates most of the OS structures supporting the process
  - checks if parent is alive:
    - If so, it holds the result value until parent requests it; in this case, process does not really die, but it enters the zombie/defunct state
    - If not, it deallocates all data structures, the process is dead
  - cleans up all waiting zombies

The wait() System Call

- A child program returns a value to the parent, so the parent must arrange to receive that value
- The `wait(&value)` system call serves this purpose:
  - it puts the parent to sleep waiting for a child’s result
  - when a child calls `exit()`, the OS unblocks the parent and returns the address of value passed by `exit()` as a result of the wait call (along with the pid of the child)
  - if there are no children alive, `wait()` returns immediately
  - also, if there are zombies waiting for their parents, `wait()` returns one of the values immediately (and deallocates the zombie)

The wait() System Call

- A child program returns a value to the parent, so the parent must arrange to receive that value
- The `wait(&value)` system call serves this purpose:
  - it puts the parent to sleep waiting for a child’s result
  - when a child calls `exit()`, the OS unblocks the parent and returns the address of value passed by `exit()` as a result of the wait call (along with the pid of the child)
  - if there are no children alive, `wait()` returns immediately
  - also, if there are zombies waiting for their parents, `wait()` returns one of the values immediately (and deallocates the zombie)

Tying all together:
the Unix shell

while(! EOF)
read input
handle regular expressions
int pid = fork()   // create child
if (pid == 0) { // child here
  exec("program", argc, argv0,...);
} else { // parent here
  ... 
}